

Inhaltsverzeichnis

1 Wiederholung und Überblick	6
1.1 Datenmodelle	6
1.1.1 Relationales Datenbankmodell	6
1.1.2 Objektorientiertes Datenbankmodell	6
1.2 Optimierer	6
1.2.1 Optimierungsarten	6
1.3 Join-Implementierungen	8
1.4 Komplexität von Operationen	8
1.5 Dateiorganisation und Zugriffspfade	8
1.5.1 Zugriffspfade	9
1.6 Transaktionen und das ACID-Prinzip	9
1.6.1 Serialisierbarkeit	10
1.7 Recovery	11
1.8 Beispiele Wiederholung und Überblick	12
2 Sichten und Datenschutz	14
2.1 Sichtenkonzept	14
2.1.1 Sichtendefinition	16
2.1.2 Sichtenimplementierung	16
2.1.3 Sichtenberechnung	17
2.1.4 Sichtenänderung	21
2.1.4.1 Kriterien	21
2.1.4.2 Kontroller der Tupelmigration	22
2.1.4.3 Problemklassifikation	23
2.1.4.4 Einschränkungen	23
2.1.5 Sichtenmaterialisierung	24
2.1.5.1 View Maintenance	24
2.1.5.2 Rekursion	24
3 Architektur von Datenbankssystemen	26
3.1 Schichtenmodell eines relationalen DBMS	26
3.1.1 Fünf-Schichtenarchitektur	26
3.1.2 Schnittstellen	26
3.1.2.1 Erläuterungen	27
3.2 Hardware und Betriebssystem	28
3.3 Pufferverwaltung	29
3.3.1 Puffer	29
3.3.1.1 Seitenzugriff	29
3.4 Speichersystem	30
3.4.1 Adressierung von Sätzen	30

3.4.2	Satztypen	31
3.5	Zugriffssystem	31
3.5.1	Indexdateien	31
3.5.2	Dateioperationen	32
3.5.2.1	Arten des Lookups	32
3.5.3	Zugriff auf Datensätze	32
3.5.4	Dateiorganisation und Zugriffspfade	33
3.6	Datensystem	34
4	Verwaltung des Hintergrundspeichers	35
4.1	Struktur des Hintergrundspeichers	35
4.1.1	Betriebssystemdateien	35
4.1.2	Blöcke und Seiten	36
4.1.3	Dienste	36
4.1.4	Abbildung der Datenstrukturen	36
4.1.4.1	Abbildungsvarianten	36
4.1.5	Datensätze und Blöcke	37
4.1.5.1	Blockungstechniken	38
4.2	Seiten, Sätze und Adressierung	38
4.2.1	Seiten	38
4.2.2	Adressierung der Datensätze	39
4.2.3	Seitenzugriff als Flaschenhals	39
4.2.4	Satztypen	39
4.2.4.1	Pinned Records	39
4.2.4.2	Unpinned Records	39
4.2.4.3	Sätze mit fester Länge	40
4.2.4.4	Sätze mit variabler Länge	40
4.2.4.5	Große unstrukturierte Datensätze	41
4.2.5	TID-Konzept (Adressierung)	43
4.3	Pufferverwaltung in Detail	44
4.3.1	Aufgaben der Pufferverwaltung	44
4.3.1.1	Suchverfahren	44
4.3.1.2	Speicherzuteilung	45
4.3.1.3	Seitenersetzungsstrategien	45
4.3.1.4	Schattenspeicherkonzept	48
5	Dateiorganisation und Zugriffsstrukturen	49
5.1	Klassifikation der Speicherungstechniken	49
5.1.1	Primär- vs. Sekundärschlüssel	49
5.1.2	Primär- vs. Sekundärindex	50
5.1.3	Dateiorganistaion vs. Zugriffspfad	51
5.1.4	Dünn- vs. Dichtbesetzter Index	51
5.1.5	(Nicht-)Geclusterter Index	52

5.1.6	Schlüsselzugriff vs. -Transformation	52
5.1.7	Ein-Attribut vs. Mehr-Attribut-Index	52
5.1.8	Statische vs. Dynamische Struktur	53
5.1.9	Klassifikation	54
5.1.10	Anforderungen an Speichertechniken	54
5.2	Statische Verfahren	54
5.2.1	Heap-Organisation	55
5.2.2	Sequenzielle Speicherung	55
5.2.2.1	Indexsequenzielle Dateiorganisation	56
5.2.2.2	Mehrstufiger Index	59
5.2.2.3	Indizierter nicht-sequenzieller Zugriffspfad	59
5.3	Baumverfahren	60
5.3.1	B-Bäume	60
5.3.1.1	Prinzip des B-Baums	61
5.3.1.2	Eigenschaften	61
5.3.1.3	Definition	61
5.3.1.4	Komplexität der Operationen	64
5.4	Hash-Verfahren	64
5.4.1	Grundprinzipien	64
5.4.2	Operationen und Zeitkomplexität	65
5.4.3	Statisches Hashen: Probleme	65
5.4.4	Lineares Hashen	65
5.4.4.1	Prinzip	66
5.5	Mehrdimensionale Speichertechniken	68
5.5.1	Mehrdimensionale Baumverfahren	68
5.5.1.1	Kdb-Bäume	68
5.5.2	Trennattribute	70
5.5.3	Brickwall-Darstellung	71
5.5.4	Mehrdimensionales Hashing	71
5.5.4.1	Idee	71
5.5.4.2	Veranschaulichung	72
5.5.4.3	Komplexität	72
5.5.5	Grid-Files	72
5.5.5.1	Zielsetzung	73
5.5.5.2	Aufbau eines Grid-Files	73
5.5.5.3	Operationen - Grid-Files	74
5.5.5.4	Buddy-System	74
6	Basisalgorithmen für Datenbankoperationen	75
6.1	Datenbankparamenter	75
6.2	Grundannahmen	76
6.3	Grundalgorithmen	76
6.3.1	Hauptspeicheralgorithmen	76

6.3.2	Zugriffe auf Datensätze	77
6.3.3	Externes Sortieren	78
6.4	Unäre Operationen	79
6.4.1	Operationen auf Scans	79
6.4.2	Selektion	80
6.4.3	Projektion	81
6.5	Binäre Operationen	81
6.5.1	Vereinigung	82
6.5.1.1	Join-Berechnung	82
7	Anfrageoptimierung	88
7.1	Grundprinzipien	88
7.1.1	Teilziele der Optimierung	88
7.1.2	Optimierte Auswertung	89
7.1.3	Auswertung mit Index	89
7.2	Phasen der Anfrageverarbeitung	90
7.3	Übersetzung in Relationsalgebra	91
7.3.1	Normalisierung	92
7.4	Logische Optimierung	93
7.4.1	Alegbraische Optimierung	93
7.4.2	Entfernen redundanter Informationen	94
7.4.3	Verschieben von Selektionen	94
7.4.4	Reihenfolge von Verbunden	95
7.4.5	Einfacher Optimierungsalgorithmus	95
7.5	Kostenbasierte Auswahl	97
7.5.1	Selektivitätsschätzung	98
7.5.2	Optimierer-Architektur	99
8	Transaktionen	100
8.1	Das Transaktionskonzept	100
8.1.1	Definition	100
8.1.2	Read/Write-Modell	100
8.1.3	Das ACID-Prinzip	101
8.1.4	Transaktionsverwaltung	101
8.2	Synchronisationsprobleme	102
8.3	Schedules und Serialisierbarkeit	103
8.3.1	Definition: Schedule	103
8.3.2	Serialisierbarkeitsbegriff	103
8.3.2.1	Ergebnisäquivalenz	104
8.3.3	Konfliktäquivalenz und -serialisierbarkeit	104
8.4	Serialisierbarkeitstest	104
8.5	Synchronisation mit Sperrverfahren	105
8.5.1	Synchronisationsprotokolle	105

8.5.2	Typen von Sperren	105
8.6	Zweiphasensperren (2PL	106
8.6.1	2PL-Varianten	107
8.6.2	Verklemmungen (Deadlocks)	107
9	Recovery und Datensicherung	108
9.1	Fehlerklassifikationen	108
9.1.1	Transaktionsfehler	108
9.1.2	Systemfehler	108
9.1.2.1	Szenario: Systemfehler	109
9.1.3	Mediafehler	109
9.2	Recovery-Klassen	110
9.3	Protokollierungsarten	110
9.3.1	Log-Buch	110
9.3.2	Physisches Protokollieren	111
9.3.3	Logisches Protokollieren	112
9.3.4	Das WAL - Prinzip	113
9.4	Recovery-Strategien	113
9.4.1	Seitenersetzungsstrategien	113
9.4.2	Propagierungsstrategien	113
9.4.3	Einbringstrategien	114
9.4.4	Konkrete Recovery-Strategien	114

Anmerkung

Alle Bilder und Inhalte sind aus den Vorlesungsunterlagen von Herrn Prof. Dr. Martin Staudt, wenn nicht anders vermerkt. Diese Unterlagen stehen nur Studenten zur Verfügung, die seinen Kurs Datenbanksysteme II im SS2016 bei ihm gehört haben.

1 Wiederholung und Überblick

1.1 Datenmodelle

1.1.1 Relationales Datenbankmodell

Dabei sind die Daten in Tabellen organisiert.

1.1.2 Objektorientiertes Datenbankmodell

Modellierte Daten, objektorientiert durch in Klassen organisierte, verzweigte Objekte.

1.2 Optimierer

- Äquivalenz von Algebra-Termen
 1. $\sigma_{A=Konst} (\text{REL1} \bowtie \text{REL2})$ und A aus REL1
 2. $\sigma_{A=Konst} (\text{REL1}) \bowtie \text{REL2}$
- allgemeine Strategie: **Selektion möglichst früh, da sie Tupelzahlen in Relationen verkleinern.**

Bsp: REL1 100 Tupel, REL2 50 Tupel, intern sind Tupel sequenziell abgelegt.

1. $5000(\bowtie) + 5000 (\sigma) = 10000$ Operationen
2. $100 (\sigma) + 10 * 50(\bowtie) = 6000$ Operationen
falls 10 Tupel in REL1 die Bedingung $A = \text{Konst}$ erfüllen

1.2.1 Optimierungsarten

- **Logische Optimierung:**
 - nutzt nur die algebraischen Eigenschaften der Operationen
 - *keine* Informationen über die Speicherstrukturen und Zugriffspfade
 - Verwendung heuristischer Regeln anstelle exakter Optimierung
 - Beispiele:
 - * Entfernung redundanter Operationen
 - * Verschieben von Operationen derart, dass Selektionen möglichst früh ausgeführt werden.
 - ~~ *algebraische Optimierung*
- **Interne Optimierung:**
 - Nutzung von Informationen über vorhandene Speicherstrukturen

- Auswahl der Implementierungsstrategie einzelner Operationen (Merge Join vs. Nested-Loop Join)
- Beispiele:
 - * Verbundreihenfolge anhand der Größe und Unterstützung der Relationen durch Zugriffspfade
 - * Reihenfolge von Selektionen nach der Selektivität von Attributen und dem Vorhandensein von Zugriffspfaden

Beispiel Algebraische Optimierung

- Entfernen *redundanter* Operationen ($r \bowtie r = r$)

$$r(BuchLangeWeg) = r(Buch) \bowtie \Pi_{ISBN, Datum}(\dots \sigma_{Datum < '31.12.1990'}(r(Ausleihe))) \quad (1)$$

- Anfrage an Sicht:

$$\Pi_{Titel}(r(Buch) \bowtie r(BuchLangeWeg)) \quad (2)$$

- Einsetzen der Sichtendefinition:

$$\Pi_{Titel}(r(Buch) \bowtie r(Buch) \bowtie \Pi_{\dots}(\dots)) \quad (3)$$

- Verschieben von Selektionen

$$\sigma_{Autor='Vossen'}(r(Buch) \bowtie \Pi_{ISBN, Datum}(\dots)) \quad (4)$$

Verbund auf kleinere Zwischenergebnisse, somit werden vorab schon mal Tupel bei der Anfrage entfernt.

$$(\sigma_{Autor='Vossen'}(r(Buch)) \bowtie \Pi_{ISBN, Datum}(\dots)) \quad (5)$$

- Reihenfolge von Verbunden

$$(r(Verlag) \bowtie r(Ausleihe)) \bowtie r(Buch) \quad (6)$$

Nachteil: erster V und entartet zum kartesischen Produkt, da keine gemeinsamen Attribute

$$r(Verlag) \bowtie (r(Ausleihe) \bowtie r(Buch)) \quad (7)$$

1.3 Join-Implementierungen

- **Merge-Join:** Verbund durch Mischen von R_1 und R_2
 - Besonders effizient, wenn eine oder beide Relationen sortiert nach dem Verbund-Attribut sortiert ist. D.h. für Verbund-Attribute X muss gelten: $X := R_1 \cap R_2$
 - r_1 und r_2 werden nach X sortiert.
 - Mischen von r_1 und r_2 . D.h. beide Relationen parallel sequenziell durchlaufen und passende Paare in das Ergebnis aufnehmen.
- **Nested-Loops-Join:** Doppelte Schleife über R_1 und R_2
 - Liegt bei einer der beiden Relationen ein Zugriffspfad für X vor, dann innere Schleife durch Zugriff über diesen Zugriffspfad ersetzen.

1.4 Komplexität von Operationen

- Selektion
 - ◆ hash-basierte Zugriffsstruktur: $O(1)$
 - ◆ sequentieller Durchlauf: $O(n)$
 - ◆ in der Regel (baumbasierte Zugriffspfade): $O(\log n)$
- Verbund
 - ◆ sortiert vorliegende Tabellen: $O(n + m)$ (Merge Join)
 - ◆ sonst: bis zu $O(n * m)$ (Nested Loops Join)
- Projektion
 - ◆ vorliegender Zugriffspfad oder Projektion auf Schlüssel: $O(n)$
 - ◆ Duplikateliminierung durch Sortieren: $O(n \log n)$

Abbildung 1: Komplexität von Operationen

1.5 Dateiorganisation und Zugriffspfade

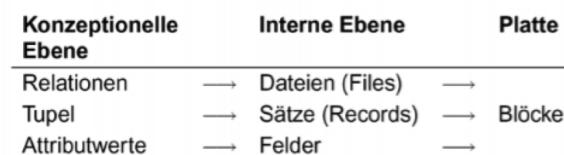


Abbildung 2: Dateiorganisation und Zugriffspfade

1.5.1 Zugriffspfade

- Primär- vs. Sekundär-Index
- sequenzielle Dateien, B-Bäume, Hashen
- eindimensional verus mehrdimensional

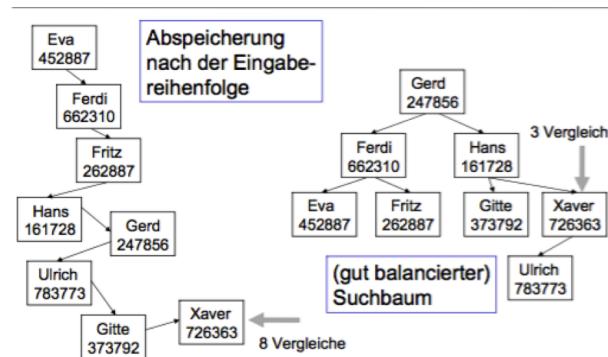


Abbildung 3: Beispiel Zugriffspfade für Telefonbuch

1.6 Transaktionen und das ACID-Prinzip

Das Ergebnis einer Transaktion soll so aussehen, als sei sie nach dem **ACID-Prinzip** abgelaufen.

- **Atomicity** (*Ununterbrechbarkeit*)
⇒ Transaktion wird ganz oder gar nicht ausgeführt
- **Consistency** (Konsistenz oder *Integritätserhaltung*)
⇒ der Zustand der von einer Transaktion hinterlassen wird genügt den Integritätsbedingungen
- **Isolation**
⇒ Bei mehrfachem Zugriff auf DB soll dieser Zugriff isoliert von allen anderen ablaufen
- **Durability** (*Dauerhaftigkeit* oder *Persistenz*)
⇒ Nach Ende einer Transaktion stehen Ergebnisse dauerhaft in der DB

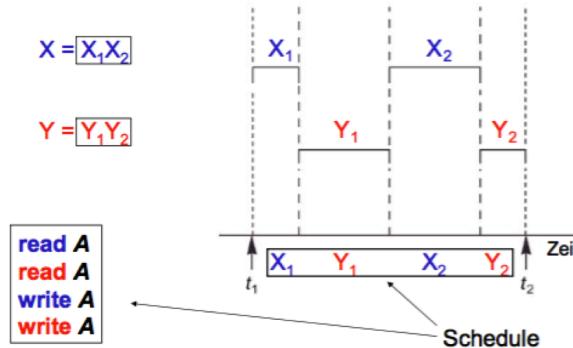


Abbildung 4: Mehrbenutzerbetrieb eines DBMS

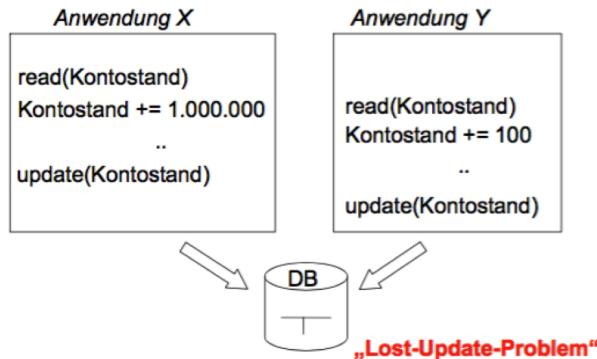


Abbildung 5: Beispiel Transaktionssynchronisation

1.6.1 Serialisierbarkeit

Ein Schedule heißt *serialisierbar*, wenn ein Ergebnis äquivalent zu dem eines serialen Schedules ist.

- Serialisierbarkeitsgraphen
- Zwei-Phasen-Sperr-Protokoll
- Zeitmarkenverfahren

Schedule S_1		Schedule S_2		Schedule S_3	
T_1	T_2	T_1	T_2	T_1	T_2
read A		read A		read A	
$A := A - 10$		read B		$A := A - 10$	
write A		$A := A - 10$		write B	
read B		$B := B - 20$		read B	
$B := B + 10$		write A		write A	
write B		read B		$B := B - 20$	
read B		read C		read B	
$B := B - 20$		$B := B + 10$		$B := B - 20$	
write B		$C := C + 20$		write B	
read C		write B		read C	
$C := C + 20$		write C		$C := C + 20$	
write C				write C	

Abbildung 6: Beispiel Serialisierbarkeit

$T_1:$ read A; $A := A - 10;$ write A; read B; $B := B + 10;$ write B;	$T_2:$ read B; $B := B - 20;$ write B; read C; $C := C + 20;$ write C;
---	---

$T_1; T_2$ und $T_2; T_1$ sind seriell

Abbildung 7: Serielle Schedules

T_1	T_2
lock A	
read A	
$A := A - 1$	
write A	
lock B	
unlock A	
	lock A
	read A
	$A := A \times 2$
	write A
	unlock A
read B	
	$\leftarrow \text{commit } T_2$
abort $T_1 \rightarrow$	$B := B/A \downarrow$

Abbildung 8: Kaskadierende Transaktionsabbrüche

1.7 Recovery

- stabiler vs. instabiler Speicher
- Log-Buch / Journal
- *Backward Recovery* : Änderungen *rückgängig* machen. $\rightsquigarrow UNDO$
- *Forward Recovery* : Änderungen nachziehen. $\rightsquigarrow REDO$
- Schattenspeicher

1.8 Beispiele Wiederholung und Überblick

```
create table Buch (
    ISBN  char(10),
    Titel  varchar(200),
    Verlagsname  varchar(30),
    primary key (ISBN),
    foreign key (Verlagsname)
        references Verlage (Verlagsname) )
```

Abbildung 9: Beispiel SQL-DDL (Data Definition Language)

Ausleih		InventarNr	Name	
		4711	Meyer	
		1201	Schulz	
		0007	Müller	
		4712	Meyer	
Buch	InventarNr	Titel	ISBN	AUTOR
	0007	Dr. No	3-125	James Bond
	1201	Objektbanken	3-111	Heuer
	4711	Datenbanken	3-765	Vossen
	4712	Datenbanken	3-891	Ullman
	4717	Pascal	3-999	Wirth

Abbildung 10: Beispiel relationales Datenbankmodell

$$\begin{aligned} & \sigma_{\text{Name}='Meyer'}(r(\text{Ausleih})) \\ & \pi_{\text{Titel}}(r(\text{Buch})) \\ & \pi_{\text{InventarNr}, \text{Titel}}(r(\text{Buch})) \bowtie \sigma_{\text{Name}='Meyer'}(r(\text{Ausleih})) \end{aligned}$$

Abbildung 11: Beispiel Relationsalgebra

```
select Buch.InventarNr, Titel, Name
from Buch, Ausleih
where Name = 'Meyer' and
      Buch.InventarNr = Ausleih.InventarNr

update Angestellte
set Gehalt = Gehalt + 1000
where Gehalt < 5000

insert into Buch values
(4867,'Wissensbanken', '3-876','Karajan')

insert into Kunde
( select LName, LAdr, 0 from Lieferant )
```

Abbildung 12: Beispiel SQL-DML (Data Manipulation Language)

```
create view Meyers as
select Buch.InventarNr, Titel, Name
from Buch, Ausleih
where Name = 'Meyer' and
      Buch.InventarNr = Ausleih.InventarNr
```

Abbildung 13: Beispiel SQL zum Erstellen einer Sicht

2 Sichten und Datenschutz

STUDENTEN				BEWERTUNGEN			
SID	VORNAME	NACHNAME	EMAIL	SID	ATYP	ANR	PUNKTE
101	Lisa	Weiss	...	101	H	1	10
102	Michael	Grau	NULL	101	H	2	8
103	Daniel	Sommer	...	101	Z	1	12
104	Iris	Winter	...	102	H	1	9

AUFGABEN			
ATYP	ANR	THEMA	MAXPT
H	1	ER	10
H	2	SQL	10
Z	1	SQL	14

Abbildung 14: Beispiel: Datenbank

2.1 Sichtenkonzept

Eine Sicht ist eine *virtuelle Realation* (bzw. virtuelle Datenbankobjekte in anderen Datenmodellen)

- Sichten sind externe DB-Schemata nach der 3-Ebenen-Schemaarchitektur.
- Sichten erlauben es, eine Anfrage auf eine Datenbank abzuspeichern.

```
CREATE VIEW HA(VORNAME, NACHNAME, GP)
AS  SELECT VORNAME, NACHNAME, SUM(PUNKTE)
    FROM STUDENTEN S, BEWERTUNGEN B
   WHERE S.SID = B.SID AND B.ATYP = 'H'
  GROUP BY VORNAME, NACHNAME, S.SID
```

Abbildung 15: Definition einer Sicht in SQL

- Das Ergebnis der SQL-Anfrage ist eine Tabelle.
- Sichten sind abgeleitete, virtuelle Tabellen, die aus den (tatsächlich abgespeicherten) Basistabellen berechnet werden.
Bei Sichten wird die definierte Anfrage gespeichert, bei Basistabellen die Tupel
- Sichten können somit nie Informationen enthalten, die nicht schon in den Basistabellen vorhanden sind.
- Eine Sicht kann aber Informationen die in den Basistabellen enthalten sind in einer anderen Form bzw. anderer strukturiert anzeigen.

- Eine Sicht unterscheidet sich wesentlich von einer mit dem Anfrageergebnis gefüllten Tabelle

```
CREATE TABLE HA2(...);
INSERT INTO HA2(VORNAME, NACHNAME, GP)
    SELECT VORNAME, NACHNAME, SUM(POINTS)
        FROM STUDENTEN S, BEWERTUNGEN B
        WHERE S.SID = B.SID AND B.ATYP = 'H'
        GROUP BY VORNAME, NACHNAME, S.SID
```

Abbildung 16: Erzeugen einer Tabelle und EInfügen von Daten in SQL

- Diese wertet die Anfrage nur einmal aus und speichert das Ergebnis dauerhaft in eine neue Tabelle.
- Eine Anfrage bei einer Sicht wird jedesmal ausgewertet, wenn die Sicht benutzt wird.

- Vorteile
 - Vereinfachung von Anfragen
 - Strukturierung der DB
 - logische Datenunabhängigkeit (Sichten stabil bei Änderungen der Datenbankstruktur)
 - Beschränkung von Zugriffen (Datenschutz)
- Probleme
 - automatische Anfragetransformation
 - Durchführung von Änderungen auf Sichten
 - Wartung von (materialisierten) Sichten
- Anwendungen
 - Bequemlichkeit / Wiederverwendung : Wiederkehrende Anfragemuster sind bereits vordefiniert.
 - Anpassung des DB-Schemas an die Wünche verschiedener Benutzer / Benutzer-Gruppen
 - Sicherheit / Datenschutz: Manche Benutzer sollen nur Teile einer Tabelle sehen können.
 - Logische Datenunabhängigkeit : Man kann neue Attribute zu einer Tabelle hinzufügen und die alte Version noch als Sicht zur Verfügung stellen.

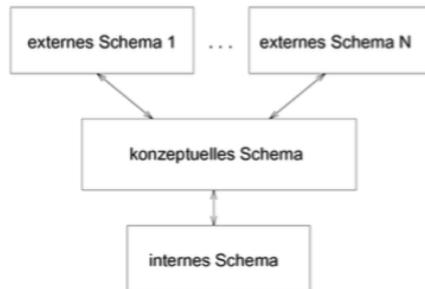


Abbildung 17: Drei-Ebenen-Schemaarchitektur

2.1.1 Sichtendefinition

Angegeben werden muss:

- *Relationenschema* (implizit oder explizit aus Ergebnistyp)
- *Berechnungsvorschrift* (Anfrage) für virtuelle Relation

```
create view SichtName [ SchemaDeklaration ]
as SQLAnfrage
[ with check option ]
```

Abbildung 18: Sichtendefinition in SQL

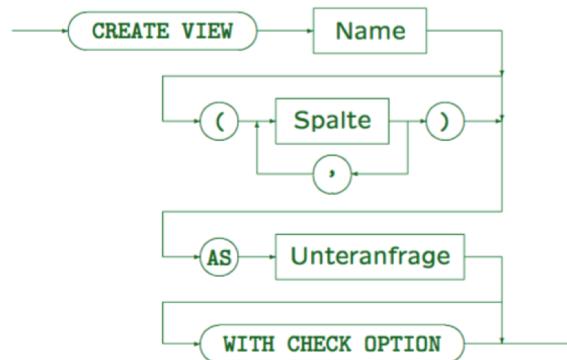


Abbildung 19: Syntax einer Sicht

- Sichten können mit dem *DROP VIEW* Befehl auch gelöscht werden.

2.1.2 Sichtenimplementierung

- Sichten sind Tabellen, die aus anderen Tabellen abgeleitet sind, die u.U. selbst wieder Sichten sind

→ Auf diese Art können komplexe Anfragen Schritt für Schritt aufgebaut werden (Rekursion).

- Die Ableitungsvorschrift ist durch die Anfrage gegeben. (Sichtendefinition)
- In der Regel sind Sichten virtuell, d.h. die Tupel einer Sicht sind nicht abgespeichert
- Anfragen über Sichten sind handhabbar wie Anfragen auf Basisrelationen

```
SELECT X.VORNAME, X.NACHNAME
FROM HA X
WHERE X.GP > 15
```

Abbildung 20: Anfrage an eine Sicht in SQL

- Die Anfrage aus der Sichtendefinition muss dann zunächst ausgerechnet und temporär als Zwischenrelation abgelegt werden

```
SELECT X.VORNAME, X.NACHNAME
FROM (SELECT VORNAME, NACHNAME, SUM(PUNKTE) GP
      FROM STUDENTEN S, BEWERTUNGEN B
     WHERE S.SID = B.SID AND B.ATYP = 'H'
   GROUP BY VORNAME, NACHNAME, S.SID) X
WHERE X.GP > 15
```

Abbildung 21: Anfrage an eine Sicht - ausgeschrieben - in SQL

- Die Zwischenrelation geht dann wie die anderen Relationen aus der FROM-Klausel in den Anfragebaum der gestellten Anfrage ein

2.1.3 Sichtenberechnung

- Die einfache Zwischenberechnung von Sichten und anschließende Verwendung bei der Anfrageverarbeitung ist meist ineffizient!
- Mit der Strategie der **Query Modification** wird eine Anfrage mit der Sichtendefinition zu einer erweiterten Anfrage verschmolzen und dann insgesamt optimiert und ausgewertet.
- Für rekursive Anfragen benötigt der Query Processor eines DBMS einen Algorithmus für die Implementierung der Rekursion, meist genügt ein Verfahren für die *transitive Hülle*

Beispielszenario

$MGA(Mitarbeiter, Gehalt, Abteilung)$ (8)

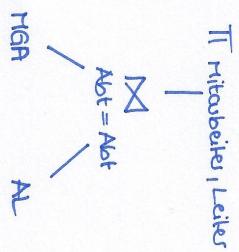
$AL(Abteilung, Leiter)$ (9)

- MGA speichert Daten über Zugehörigkeit von Mitarbeitern zu Abteilung und deren jeweiliges Gehalt
- AL gibt für jede Abteilung den Abteilungsleiter an

hasBoss

Create view hasBoss

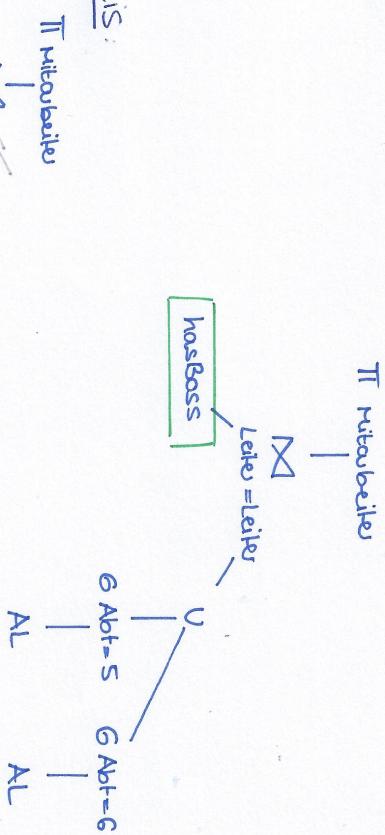
as
SELECT Mitarbeiter, Leiter
FROM HGA, AL
WHERE HGA.Abtl = AL.Abt



Anfrage

SELECT Mitarbeiter
FROM hasBoss, AL
WHERE hasBoss.Leiter = AL.Leiter
AND Abtl = 5 OR Abtl = 6

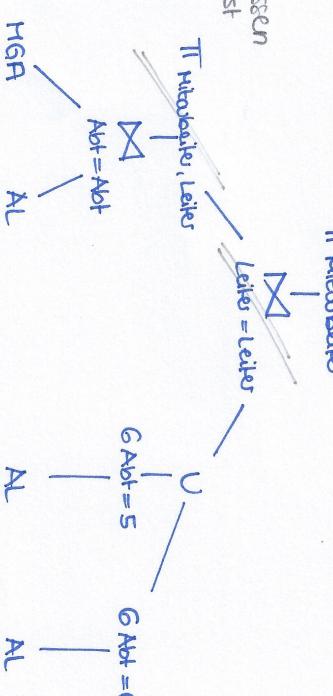
Ergebnis:



\Rightarrow

Join über Leiter kann weggelassen werden, da diese nur nötig ist um die Vereinigung mit dem Abt zu machen

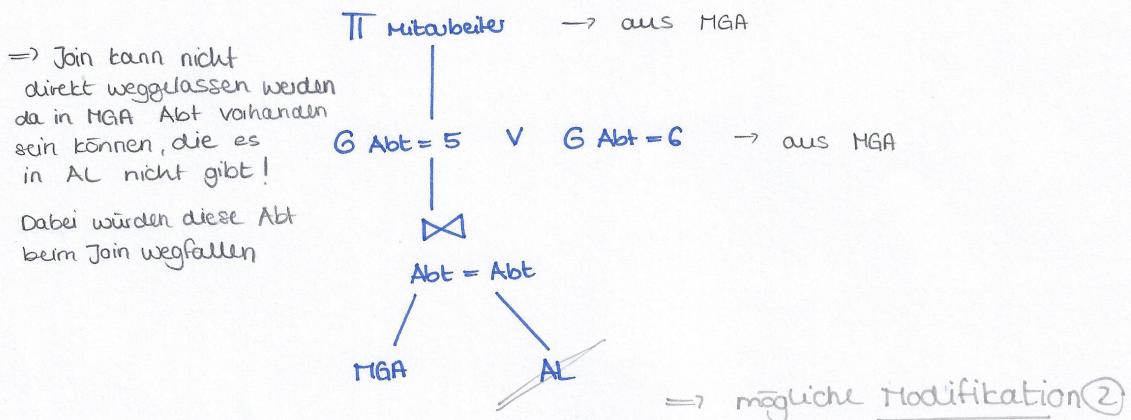
\rightarrow Infos in HGA schon enthalten



\Rightarrow mögliche Modifikationen ①

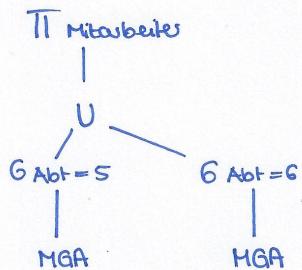
\rightarrow dabei müssen Blätter von unten nach oben transportiert werden

Nach ① Modifikation:



=> AL kann wegfallen, wenn es eine referentielle Integritätsbedingung
FOREIGN KEY MGA.Abt → AL.Abt

Nach ② Modifikation:



SQL:

```
SELECT Mitarbeiter
FROM MGA
WHERE Abt = 5 OR Abt = 6
```

=> Somit spart man zwei Join-Bedingungen

2.1.4 Sichtenänderung

2.1.4.1 Kriterien

Effektkonformität Benutzer sieht Effekt als wäre die Änderung af der Sichtenrelation direkt ausgeführt worden.

Minimalität Basisdatenbank sollte nr minimal geändert werden, um den erwähnten Effekt zu erhalten.

Konsistenzerhaltung Änderungen einer Sicht darf zu keinen Integritätsverletzungen der Basisdatenbank führen.

Respektierung des Datenschutzes Wird die Sicht aus Datenschutzgründen eingeführt, darf der bewusst ausgeblendete Teil von Änderungen durch die Sicht nicht betroffen sein.

Nach syntaktischer Transformation:

```
select Abteilung
from MGA
where sum(Gehalt) > 500
group by Abteilung
```

Keine syntaktische korrekte SQL-Anfrage! Korrekt wäre:

```
select Abteilung
from MGA
group by Abteilung
having sum(Gehalt) > 500
```

Abbildung 22: Beispiel: Projektionssicht

```
MG :=  $\sigma_{Gehalt > 20}(\pi_{Mitarbeiter, Gehalt}(MGA))$ 

create view MG as
select Mitarbeiter, Gehalt
from MGA
where Gehalt > 20

Tupelmigration: Ein Tupel MGA('Zuse', 25, 'Info'), wird aus
der Sicht „herausbewegt“:
update MG set Gehalt = 15
where Mitarbeiter = 'Zuse'
```

Abbildung 23: Beispiel: Selektionssicht

2.1.4.2 Kontroller der Tupelmigration

```
create view MG as
select Mitarbeiter, Gehalt
from MGA
where Gehalt > 20
with check option
```

Abbildung 24: Erzeugung einer view

MGAL := MGA \bowtie AL

In SQL:

```
create view MGAL as
select Mitarbeiter, Gehalt,
       MGA.Abteilung, Leiter
  from MGA, AL
 where MGA.Abteilung = AL.Abteilung
```

Änderungsoperationen in der Regel nicht eindeutig
übersetzbare:

```
insert into MGAL
values ('Turing', 30, 'Info', 'Zuse')
```

Abbildung 25: Beispiel: Verbundsichten - Step 1

Änderung wird transformiert zu

```
insert into MGA values ('Turing', 30, 'Info')
```

plus

1. Einfügeanweisung auf AL:

```
insert into AL values ('Info', 'Zuse')
```
2. oder alternativ:

```
update AL set Abteilung = 'Info'
           where Leiter = 'Zuse'
```

 - besser bzgl. Minimalitätsforderung
 - widerspricht aber Effektkonformität!

Abbildung 26: Beispiel: Verbundsichten - Step 2

```
create view AS (Abteilung, SummeGehalt)
as
select Abteilung, sum(Gehalt)
from MGA
group by Abteilung
```

Folgende Änderung ist nicht eindeutig umsetzbar:

```
update AS
set SummeGehalt = SummeGehalt + 1000
where Abteilung = 'Info'
```

Abbildung 27: Beispiel: Aggregierungssichten

2.1.4.3 Problemklassifikation

1. Verletzung der Schemadefinition (z.B. beim Einfügen von Nullwerten bei Projektionssicht)
2. Datenschutz: Seiteneffekte auf nicht-sichtbaren Teil der Datenbank vermeiden (Tupelmigration, Selektionssicht).
3. Nicht immer eindeutige Transformation: Auswahlproblem.
4. Aggregierungssichten (u.a.): Keine sinnvolle Transformation möglich.
5. elementare Sichtenänderung soll genau einer atomaren Änderung auf Basisrelation entsprechen: 1:1-Beziehungen zwischen Sichttupeln und Tupeln der Basisrelation (kein Herausprojizieren von Schlüssel)

2.1.4.4 Einschränkungen

- Änderbar nur *Selektions- und Projektionssichten* (Verbund und Mengenoperationen nicht erlaubt)
- 1:1 - Zuordnung von Sichttupeln zu Basisrelation: Kein **distinct** in Projektionssichten
- Arithmetik und Aggregatfunktionen im **select** Teil sind verboten
- Genau eine Referenz auf einen Relationsnamen im **from** Teil erlaubt (kein Selbsverbund)
- Keine Unteranfragen mit “Selbsbezug” im **where** Teil erlaubt
- **group by** und **having** verboten

2.1.5 Sichtenmaterialisierung

Die Extension einer Sicht wird beim erstmaligen Berechnen gespeichert (materialisiert). Später kommende Anfragen können dann ohne Berechnung darauf zugreifen.

Problem: Die Sicht spiegelt den Datenbankzustand zum Zeitpunkt der Erstberechnung wieder!

Möglichkeit 1: Die Sicht wird in bestimmten Abständen neu berechnet. Anfragen werden in der Zwischenzeit möglicherweise auf der nicht ganz aktuellen Sicht ausgewertet (relaxierte Kohärenz).

Beispiel: Data Warehouse

Möglichkeit 2: Die Sicht wird gewartet (*View Maintenance*)

2.1.5.1 View Maintenance

- **Strategie 1:** Wann immer sich in einer der Basisrelationen der Sicht etwas, wird die Sicht neu berechnet.
 - einfach zu realisieren *Trigger*
 - die stattgefundenen Änderungen spielen keine Rolle
 - die Neuberechnung kann teuer und überflüssig sein
- **Strategie 2:** (→ Differenzierung) Die konkreten Änderungen der Basisrelationen werden ausgenutzt, um Änderungen an der Sicht zu berechnen, d.h. welche Tupel hinzukommen oder wegfallen.

Die Berechnung des “Deltas” einer Sicht, d.h. der Differenz zwischen alter und neuer Sichtenextension, entspricht einer partiellen Differenzierung des Relationenabdrucks nach jeder einzelnen Basisrelation. Schwierig wird das View-Maintenance Problem durch das Auftreten von Aggregatfunktionen oder Rekursion, die dann spezielle Verfahren erfordern. Hat man Sichten auf Sichten, setzen die meisten Sichtwartungsverfahren voraus, dass alle beteiligten Sichten jeweils materialisiert sind. Es gibt auch Verfahren für extrem materialisierte (hierarchische) Sichten, die ohne Materialisierung der verwendeten Sichten und ohne Zugriff auf die “alte” Materialisierung auskommen

2.1.5.2 Rekursion

Transitive Hülle Die einfachste Form der Rekursion kann mit der Transitiven Hüller erreicht werden.

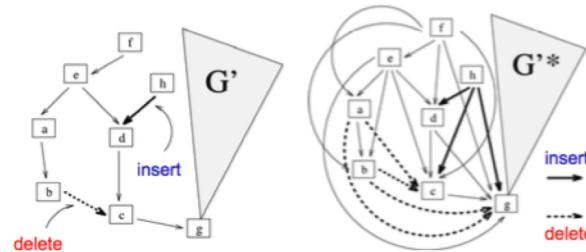


Abbildung 28: Beispiel: Transitive Hülle

```
CREATE TABLE Flug (
    Abflug VARCHAR(30),
    Ziel VARCHAR(30),
    Carrier VARCHAR(6),
    Preis Decimal(6,2))
```

Abflug	Ziel	Carrier	Preis
Frankfurt	Sydney	LH	1009
Zürich	London	BA	599
Paris	New York	AF	1299
Sydney	New York	TWA	1549
Zürich	Frankfurt	SR	499

```
WITH RECURSIVE Erreichbar(Abflug, Ziel) AS
(
    SELECT Abflug, Ziel
    FROM Flug
    UNION
    SELECT e.Abflug, f.Ziel
    FROM Erreichbar e, Flug f
    WHERE e.Ziel = f.Abflug )
SELECT * FROM Erreichbar WHERE Abflug = 'Zürich'
```

Abflug	Ziel
Zürich	London
Zürich	Frankfurt
Zürich	Sydney
Zürich	New York

Abbildung 29: Beispiel: Transitive Hülle in SQL99

Berechnung rekursiver Anfragen

- Sind Anfragen rekursiv definiert, kann die Antwort mit Hilfe des Relationsalgebra - Ausdrucks *bottom-up* aus den zugrundeliegenden Basistabellen berechnet werden.
- Die rekursiven Anfragen entsprechen Gleichungssystemen
- Man benötigt ein Iterationsverfahren, das sukzessive weitere neu hinzukommende Tupel mit Hilfe der in der letzten Iterationsstufe ermittelten Tupel berechnet.
- Die dazu verwendeten Algorithmen entsprechen den bekannten Verfahren aus anderen Bereichen z.B. der Numerik.

3 Architektur von Datenbankssystemen

3.1 Schichtenmodell eines relationalen DBMS

3.1.1 Fünf-Schichtenarchitekut

- basierend auf Idee von Senko 1973 (wurde von Härder 1987 weiterentwickelt)
- Umsetzung im Rahmen des IBM-Prototyps *System R*
- genaue Beschreibung der Transformationskomponenten
 - schrittweise Transformation von Anfragen / Änderungen bis hin zu Zugriffen auf Speichermedien
 - Definition der Schnittstellen zwischen zwei Komponenten

3.1.2 Schnittstellen

mengenorientierte Schnittstelle deklarative DML auf Tabllen, Sichten, Zeilen

satzorientierte Schnittstelle Sätze, logische Dateien, logischer Zugriff. Navigierender Zugriff.

interne Schnittstelle Sätze, Zugriffspfade. Manipulation von Sätzen und Zugriffspfaden.

Pufferschnittstelle Seiten, Seitenadressen. Freigeben und Bereitstellen.

Datei- oder Seitenschnittstelle Hole Seite, schreibe Seite.

Geräteschnittstelle Spuren, Zylinder. Armbewegungen

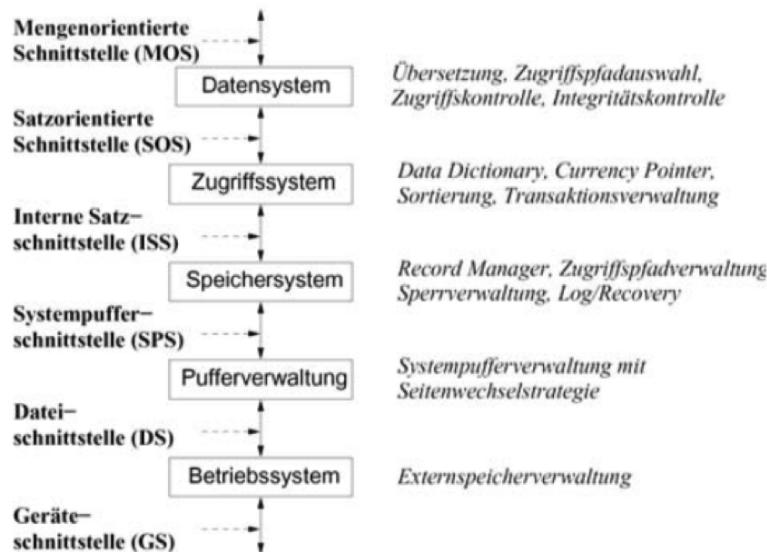


Abbildung 30: Funktionen

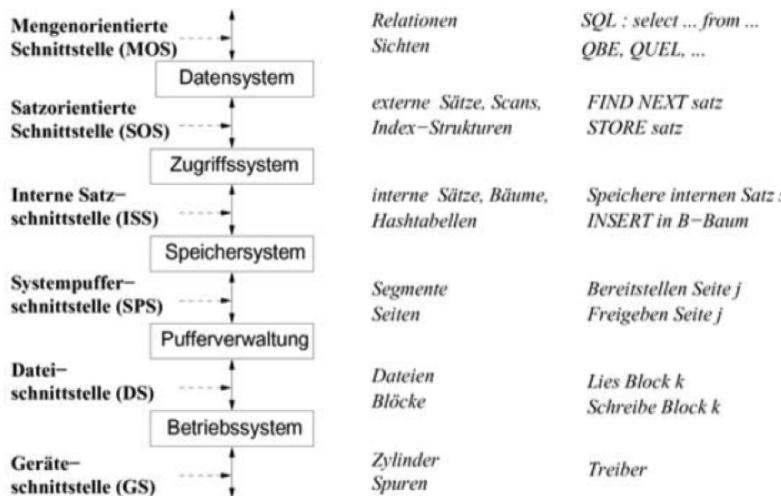


Abbildung 31: Objekte

3.1.2.1 Erläuterungen

- mengenorientierte Schnittstelle **MOS**:
 - deklarative Datenmanipulationssprache auf Tabellen und Sichten (etwa SQL)
- durch Datenystem auf satzorientierte Schnittstelle **SOS** umgesetzt:

- navigierender Zugriff auf interne Darstellung der Relationen
- maipulierte Objekte: typisierte Datensätze und interne Relationen sowie logische Zugriffspfade (Indexe)
- Aufgaben des Datensystems: Übersetzung und Optimierung von SQL-Anfragen.
- durch Zugriffssystem auf interne Satzschnittstelle **ISS** umgesetzt:
 - interne Tupel einheitlich verwalten, ohne Typisierung
 - Speicherstrukturen der Zugriffspfade (konkrete Operationen auf B-Bäume und Hash-Tabellen). Mehrbenutzerbetrieb mit Transaktionen.
- durch Speichersystem Datenstrukturen und Operationen der ISS auf interne Seiten eines virtuellen linearen Adressraums umgesetzt:
 - Manipulation des Adressraums durch Operationen der Systempufferschnittstelle **SPS**
 - Typische Objekte: interne Seiten, Seitenadressen
 - Typische Operationen: Freigabe und Bereitstellen von Seiten, Seitenwechselstrategien, Sperrverwaltung, Schreiben des Log-Buchs.
- durch Pufferverwaltung interne Seiten auf Blöcke der Dateischnittstelle **DS** abbilden
 - Umsetzung der DS-Operationen auf Geräteschnittstelle erfolgt durch BS

3.2 Hardware und Betriebssystem

Betriebssystemebenen → Grundlage für datenbankbezogene Ebenen. Es werden Treiberprogramme zum Zugriff auf die Daten von Medien und Caching-Mechanismen benötigt. Bei den Prozessoren und Rechnerarchitekturen werden klassische Industrie-Standards verwendet (aber: Datenbankmaschinen)
An das Speichermedium werden spezielle Anforderungen in Hinblick auf die Speicherhierarchie gestellt.

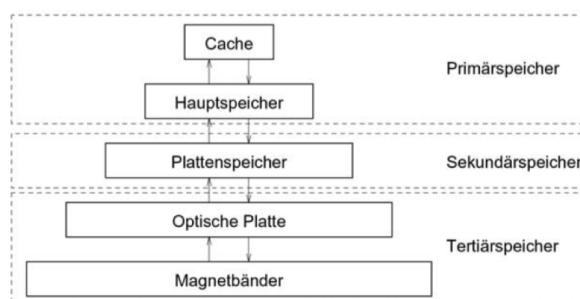


Abbildung 32: Speicherhierarchie

3.3 Pufferverwaltung

- Benötigte Blöcke des Sekundärspeichers im Hauptspeicher verwalten
- Speicherplatz für begrenzte Mengen von Seiten im Hauptspeicher: *Puffer*
- Aufgaben:
 - Verdrängen nicht mehr im Puffer benötigte Seiten (Seitenwechselstrategien)
 - Zuteilung von Speicherplatz für Seiten
 - Suchen und Erstzen von Seiten im Puffer
 - Optmierung der Lastverteilung zwischen parallelen Transaktionen
- Unterschied: Verantwortung zur Verwaltung des Puffer liegt bei Datenbankssystem
↔ Cache auf der Betriebssystemebene

3.3.1 Puffer

- Je nach Hauptspeichergröße beträchtlicher Umfang (bis zu einigen 100 MB)
- trotzdem nur ganz geringer Bruchteil der Datenbank (weniger als 1 %)
- alle Lese- und Schreibezugriffe von oder auf Seiten im Puffer ⇒ kann zum Flaschenhals werden!
- verfügbarer Hauptspeicher sehr groß und Datebank relativ klein ⇒ gesamte Datenbank in den Puffer: *Hauptspeicher-Datenbank* (engl. main memory databases)

3.3.1.1 Seitenzugriff

Die höhere Schicht (Speichersystem) fordert bei Pufferverwaltung eine Seite an (*logische Seitenreferenz*).

- angeforderte Seite im Puffer: wird dem Speichersystem zur Verfügung gestellt.
- angeforderte Seite nicht im Puffer (*page fault*): *physische Seitenreferenz* durch Pufferverwaltung an Betriebssystemebene. I.a. bei gefülltem Puffer eine Seite aus dem Puffer verdrängen. Falls die zu verdrängende Seite geändert wurde, vorher auf den Sekundärspeicher schreiben.

Aufwand pro E/A-Operation: 2500 Instruktionen in CPU; 15 bis 30 ms für Zugriff auf Sekundärspeicher.

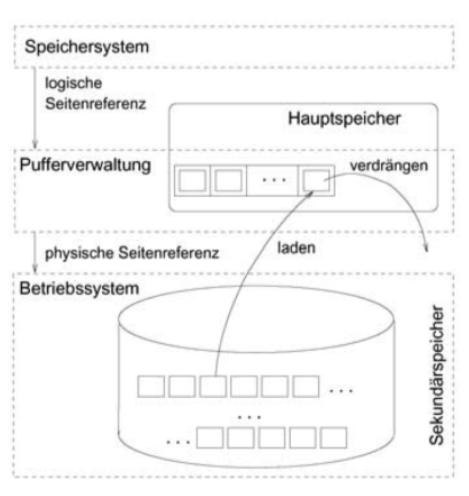


Abbildung 33: Speicherhierarchie

3.4 Speichersystem

Puffer: Seiten ↔ Speichersystem: interne Datensätze ↔ Zugriffssystem: logische Datensätze, interne Tupel

Struktur	Systemkomponente
Tupel	Datensystem
internes Tupel oder logischer Datensatz	Zugriffssystem
interner Datensatz	Speichersystem

Abbildung 34: Speicherhierarchie

- Anwendungsobjekt im Speichersystem als interne Sätze
- Hilfsdaten wie Indexeinträge als interne Sätze

3.4.1 Adressierung von Sätzen

- Problem bei Adressierung: Änderungen im Datenbestand \Rightarrow effiziente Änderung von Adressen.
- Beispiel: interne Sätze adressiert mit Offset x relativ zum Seitenanfang (interner Satz startet auf Byte x) \Rightarrow Änderungen auf dieser Seite haben Auswirkungen auf verwendete Tupeladresse.
- Besser: *TID-Konzept* (Tupel-Identifier)
 - Adresse: Seitennummer und offset in einer Liste von Tupelzeigern am Anfang der Seite

- Eintrag im Zeigerfeld bestimmt Offset des Satzes
- Ändert sich Position des internen Satzes auf der Seite \Rightarrow nur Eintrag lokal om Zeigerfeld verändern; alle “aussen” verwendete Adressen beleiben stabil.

3.4.2 Satztypen

- *Nicht-Spannsätze (unspanned records)* auf maximal eine Seite; Satz zu groß für eine in Bearbeitung befindliche Seite \Rightarrow von Freispeicher verwaltung eine neue Seite anfordern.
- *Spannsätze (spanned records)* könnnen mehrere Seiten überspannen; Satz zu groß \Rightarrow Beginn des Satzes auf deiser Seite, Überlauf auf neuer Seite speichern.
- *Sätze fester Länge* : für bestimmten Tupeltyp feste Anzahl von Bytes (bei strings alle Attributwerte mit gleicher Anzahl Bytes speichern)
- *Sätze variabler Länge* : nur die wirklich benötigte Anzahl von Bytes speichern (Anzahl der Bytes pro Datensatz variabel)

3.5 Zugriffssystem

- Zugriffssystem abstahiert von interner Darstellung der Datensätze auf der Seite
- *logische Datensätze, interne Tupel*
- interne Tupel können Elemente einer Dateidarstellung der konzeptuellen Relation oder Elemente eines Zugriffspfads auf die konzeptuellen Relationen sein.
- interne Tupel bestehen aus Feldern (entsprechen Attributen bei konzeptuellen Tupeln)
- Operationen im Zugriffspfad sind typischerweise *Scans* (interne Curser auf Dateien oder Zugriffspfaden)

3.5.1 Indexdateien

- Zugriffspfad auf einer Datei selbst wieder Datei: *Indexdatei*
- Index enthält neben den Attributwerten der konzeptuellen Relation, über die ein schneller Zugriff auf Relation verwirklicht werden soll, eine Liste von Tupeladressen.
- zugeordnete Adressen verweisen auf Tupel, die den indizierten Attributwert beinhalten
- Index über Primärschlüssel \Rightarrow Liste der Tupeladressen einelementig: *Primärindex*

- Index über beliebige andere Attributmenge: *Sekundärschlüssel* (obwohl Attributwerte gerade **keine** Schlüsseleigenschaften besitzen müssen)
- Index über Sekundärschlüssel: *Sekundärindex*

3.5.2 Dateioperationen

- Einfügen eines Datensatzes **insert**
- Löschen eines Datensatzes **remove** oder **delete**
- Modifizieren eines Datensatzes **modify**
- Suchen und Finden eines Satzes **lookup** oder **fetch**

3.5.2.1 Arten des Lookups

- Gegebenen Attributwert für bestimmtes Feld, gesucht interne Tupel, die diesen Attributwert besitzen: *single-match query*
- Gegebene Wertekombination für bestimmte Feldkombination, gesucht alle Tupel, die diese Attributwerte besitzen:
 - Werte für alle Felder der (Index-) Datei: *exact-match query* (*single-match query ist einfacher Spezialfall*)
 - Werte für einige Felder der (Index-) Datei: *partial-match query*
- Gegebenes Wertebereich für ein oder mehrere Attribute, gesucht alle internen Tupel, die Attributwerte in diesem Intervall besitzen: *range query*

3.5.3 Zugriff auf Datensätze

- Datensätze in Abhängigkeit von Primärschlüsselwert in einer Datei
 - geordnet oder
 - gehasht (gestreut)gespeichert ⇒ schneller Zugriff über Primärschlüssel
- schneller Zugriff über andere Attributmengen (Sekundärschlüssel) standardmäßig über Indexdateien realisiert.

3.5.4 Dateiorganisation und Zugriffspfade

Primärschlüssel / Sekundärindex

- Primärschlüssel-Zugriff (nur eine Tupeladresse pro Attributwert)
- Sekundärschlüssel-Zugriff (mehrere Tupeladressen pro Attributwert möglich)
- oft: Primärschlüssel bestimmt Dateiorganisationsform, Sekundärschlüssel bestimmt Zugriffspfade (Indexdateien)

eindimensional / mehrdimensional

- Unterstützung des Zugriffs für feste Feldkombination (*exact-match*)
- Unterstützung des Zugriffs für variable Feldkombination (*partial-match*)

statisch / dynamisch

- statische Dateiorganisationsform oder Zugriffspfad nur optimal bei einer bestimmten Anzahl von zu verwaltenden Datensätzen.
- dynamische Dateiorganisationsformen oder Zugriffspfade unabhängig von der Anzahl der Datensätze (automatische, effiziente Anpassung an wachsende oder schrumpfende Datenmengen)

Beispiele

- B-Bäume
 - dynamischer, eindimensionaler Zugriffspfad
 - in den meisten Datenbanksystemen über mehrere Attribute einer Datei definierbar
 - aber auch ein *exact-match* auf dieser Feldkombinatiton möglich
- klassisches Hash-Verfahren
 - statische, eindimensionale Dateiorganisationsform
 - bei wachsenden Tupelmengen immer mehr Kollisionen zu erwarten

3.6 Datensystem

- *Optimierung:* mengenorientierte Anfrage (SQL) muss durch System optimiert werden
 - Umformung des Anfrageausdrucks in einen effizienter zu bearbeitenden Ausdruck (*Query Rewriting, Konzeptuelle oder Logische Optimierung*)
 - Auswahl des effizientesten Anfrageausdrucks nach Kostenschätzung
 - *Interne Optimierung:* Auswahl der zur Anfragebearbeitung sinnvollen Zugriffspfade und Auswertungsalgorithmen für jeden relationalen algebraischen Operator
- *Zugriffspfadauswahl:* bestimmt die internen Strukturen, die bei Abarbeitung einer Anfrage benutzt werden sollen
- *Auswertung:* Wahl der Auswertungsalgorithmen kann Antwortzeit auf eine Anfrage entscheidend beeinflussen; im Zusammenspiel mit der Zugriffspfadauswahl muss Datensystem die Auswertungsalgorithmen auswählen

4 Verwaltung des Hintergrundspeichers

4.1 Struktur des Hintergrundspeichers

Abstraktion von Speicherungs- oder Sicherungsmedium
Modell: Folge von Blöcken

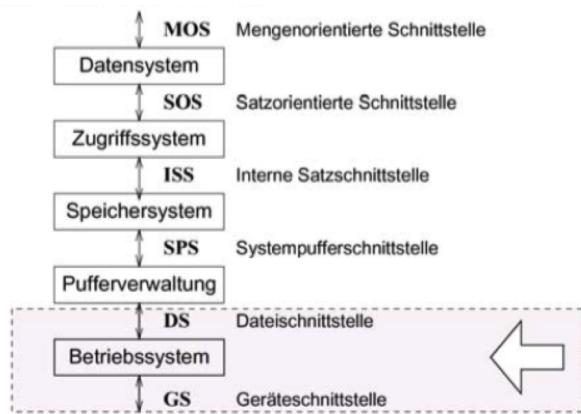


Abbildung 35: Einordnung in Architektur

4.1.1 Betriebssystemdateien

- Jeder Relation oder jeder Zugriffspfad in genau einer Betriebssystem-Datei
- Ein oder mehrere BS-Dateien, DBS verwaltet Relationen und Zugriffspfade selbst innerhalb dieser Dateien
- DBS steuert selbst Magnetplatte an und arbeitet mit den Blöcken in ihrer Ursprungsform (*raw device*)

Warum nicht immer BS-Dateiverwaltung?

- Betriebssystemunabhängigkeit
- in 32-Bit-Betriebssystemen: Dateigröße maximal 4GB
- BS-Dateien auf maximal einem Medium
- betriebssystemseitige Pufferverwaltung von Blöcken des Sekundärspeichers im Hauptspeicher genügt nicht den Anforderungen des DBS.

4.1.2 Blöcke und Seiten

- Zuordnung der physischen Blöcke zu *Seiten*
- meist mit festen Faktoren: 1, 2, 4 oder 8 Blöcke einer Spur auf einer Seite
- hier: “Ein Block - eine Seite”
- höhere Schichten des DBS adressieren über Seitennummer

4.1.3 Dienste

- Allokation oder Deallocatation von Speicherplatz
- Holen und Speichern von Seiteninhalten
- Allokation möglichst so, dass logisch aufeinanderfolgende Datenbereiche (etwa einer Relation) auch möglichst in aufeinanderfolgende Blöcke der Platte gespeichert werden.
- Nach vielen Update-Operationen: *Reorganisationsmethoden*
- *Freispeicher verwaltung*: doppelt verkettete Liste von Seiten

4.1.4 Abbildung der Datenstrukturen

- Abbildung der konzeptuellen Ebene auf interne Datenstrukturen
- Unterstützung durch *Metadaten* (im Data Dictionary, etwa das interne Schema)

Konz. Ebene	Interne Ebene	Dateisystem/Platte
Relationen →	Log. Dateien →	Phys. Dateien
Tupel →	Datensätze →	Seiten/Blöcke
Attributwerte →	Felder →	Bytes

Abbildung 36: Abbildung der Datenstrukturen

4.1.4.1 Abbildungsvarianten

- Beispiel 1: jede Relation in je einer Datei, diese insgesamt in einer einzigen physischen Datei.

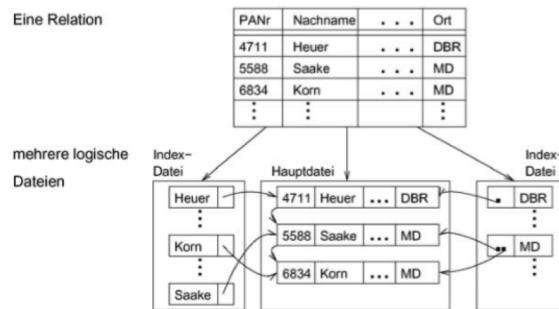


Abbildung 37: Abbildung der Datenstrukturen

- Beispiel 2: Cluster-Speicherung - mehrere Relationen in einer logischen Datei

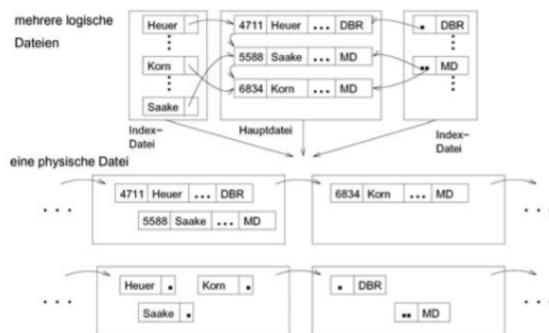


Abbildung 38: Abbildung der Datenstrukturen

4.1.5 Datensätze und Blöcke

- Datensätze (evtl. variabler Länge) in die aus einer fest vorgeschriebenen Anzahl von Bytes bestehende Blöcke einpassen: *Blöcken*
- Blöcken abhängig von variabler oder fester Feldlänge der Datenfelder
 - Datensätze mit variabler Satzlänge: höherer Verwaltungsaufwand beim Lesen und Schreiben, Satzlänge immer wieder neu ermitteln.
 - Datensätze mit fester Satzlänge: höherer Speicheraufwand.

4.1.5.1 Blockungstechniken

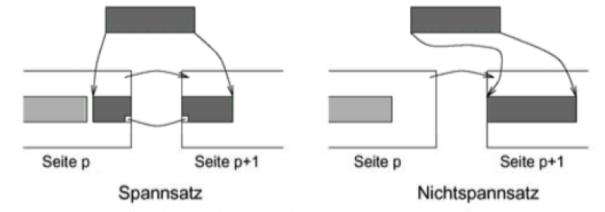


Abbildung 39: Blockungstechniken

- *Nichtspannsätze*: jeder Datensatz in maximal einem Block
- *Spannsätze*: Datensatz eventuell in mehreren Blöcken
- Standard: Nichtspannsätze (nur im Falle von BLOBs oder CLOBs Spannsätze üblich)

4.2 Seiten, Sätze und Adressierung

Struktur der Seiten: doppelt verkettete Liste. Die freien Seiten liegen in der *Freispeicher-verwaltung*

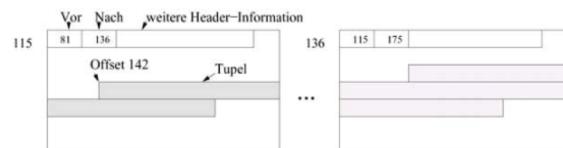


Abbildung 40: Seiten, Sätze und Adressierung

4.2.1 Seiten

- *Header*
 - Informationen über Vorgänger- und Nachfolger-Seiten
 - eventuell auch Nummer der Seite selbst
 - Informationen über Typ der Sätze
 - freier Platz
- *Datensätze*
- unbelegte Bytes

4.2.2 Adressierung der Datensätze

- adressierbare Einheiten:
 - Zylinder
 - Spuren
 - Selektoren
 - Blöcke oder Seiten
 - Datensätze in Blöcken oder Seiten
 - Datenfelder in Datensätzen
- Beispiel: Adresse eines Satzes durch Seitennummer und Offset (relative Adresse in Bytes vom Seitenanfang)

4.2.3 Seitenzugriff als Flaschenhals

- Maß für die Geschwindigkeit von Datenbankoperationen Anzahl der Seitenzugriffe auf dem Sekundärspeicher (wegen Zugriffslücke)
- Faustregel: Geschwindigkeit des Zugriffs \Leftarrow Qualität des Zugriffspfads \Leftarrow Anzahl benötigter Seitenzugriffe
- Hauptspeicheroperationen nicht beliebig vernachlässigbar

4.2.4 Satztypen

4.2.4.1 Pinned Records

- *Fixierte Datensätze* sind an ihre Position gebunden.
- Beispiel: Verwiese mit Zeigern aus anderer, schwer auffindbarer Seite, etwa Verschiebung des Datensatzes.
- Gefahr: ins Leere verweisender Zeiger (*dangling pointers*)

4.2.4.2 Unpinned Records

- *Unfixierte Datensätze* verweisen mit “logischem Zeiger”
- etwa: Matrikelnummer bei Studenten-Datensatz
- diesen an zentraler Stelle (etwa auf der Indexdatei zur Studenten-Relation) auf aktuelle Adresse umsetzen
- Nachteil: Verschieben des Datensatzes benötigt neben dem Laden der beiden direkt betroffenen Seiten (Quell und Zielseite der Verschiebungsoperation) auch noch Holen der Indexseite
⇒ behoben mit dem TID-Konzept

4.2.4.3 Sätze mit fester Länge

SQL: Datentypen fester und variabler Länge

- $char(n)$ Zeichen der festen Länge n
- $varchar(n)$ Zeichenkette fester Länge mit der maximalen Länge n

Aufbau der Datensätze, falls Datenfelder feste Länge:

1. *Verwaltungsblock* mit
 - Typ eines Satzes (wenn unterschiedliche Satztypen auf einer Seite möglich)
 - Löschbit
2. *Freiraum* zur Justierung des Offsets
3. *Nutzdaten* des Datensatzes

4.2.4.4 Sätze mit variabler Länge

- Im Verwaltungsblock nötig: Satzlänge l , um die Länge des Nutzdatenbereichs d zu kennen:

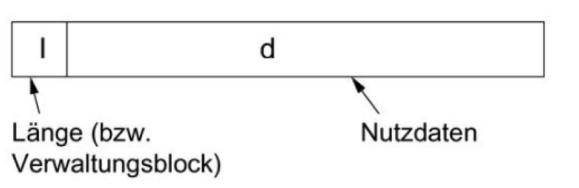


Abbildung 41: Sätze mit variabler Länge

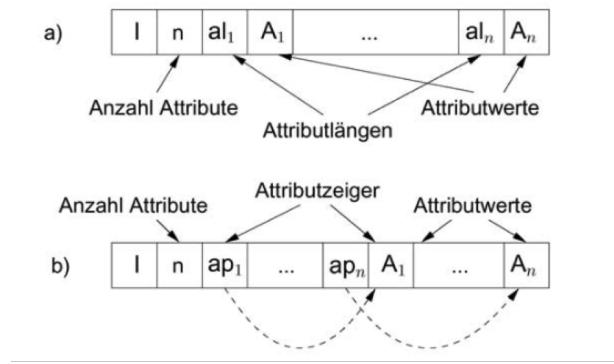


Abbildung 42: Sätze mit variabler Länge

Speicherung bei variabler Länge

Strategie a:

Jedes Datenfeld variabler Länge A_i beginnt mit einem Längenzeiger al_i , dieser gibt an, wie lang das folgende Datenfeld ist.

Strategie b:

Am Beginn des Satzes wird nach dem Satz-Längenzeiger l und der Anzahl der Attribute ein Zeigerfeld $ap_1 \dots ap_n$ für alle variabel langen Datenfelder eingerichtet.

Vorteil dieser Strategie b) ist eine leichtere Navigation innerhalb des Satzes (auch für Sätze in Seiten \Rightarrow TID)

Anwendung variabel langer Datenfelder “Wiederholungsgruppen”: Liste von Werten des gleichen Datentyps

- Zeichenketten variabler Länge wie $varchar(n)$ sind Wiederholungsgruppen mit $char$ als Basisdatentyp. Mathematisch also eine Kleenesche Hülle $(char)^*$)
- Mengen- oder listenwertige Attributwerte, die im Datensatz selbst denormalisiert gespeichert werden sollen (Speicherung als geschachtelte Relation oder Cluster-Speicherung), bei einer Liste von $integer$ -Werten wäre dies $((integer)^*)$
- Adressfelder für die Indexdatei, die zu einem Attributwert auf mehreren Datensätzen zeigt (*Sekundärindex*), also $(pointer)^*$

4.2.4.5 Große unstrukturierte Datensätze

- RDBS-Datensätze für sehr große, unstrukturierte Informationen
 - *Binary Large Objects (BLOBs)*: Bytefolgen wie Bilder, Audio- und Videosequenzen
 - *Character Large Objects (CLOBs)*: Folgen von ASCII-Zeichen (unstrukturierter ASCII-Text)
- lange Felder überschreiten i.a. Grenzen einer Seite, deshalb nur Nicht-BLOB-Felder auf der Orginalseite speichern!

BLOB-Speicherung Lösung 1:

- Als Attributwert Zeiger: Zeiger zeigt auf Begin einer Seite- oder Blockliste, die BLOB aufnimmt

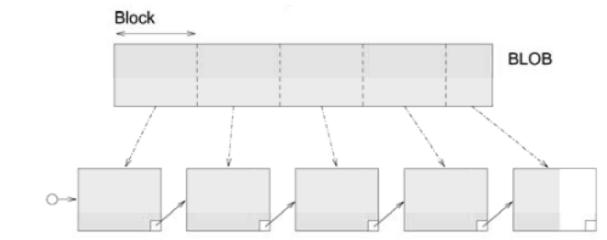


Abbildung 43: BLOB-Speicherung - Lösung 1

- Vorteil beim Einfügen, Löschen und Modifizieren
- Nachteil bei wahlfreien Zugriff in das BLOB hinein

Lösung 2:

- Als Attributwert BLOB-Directory:
 - BLOB-Größe
 - weitere Verwaltungsinformationen
 - mehrere Zeiger, die auf die einzelnen Seiten verweisen

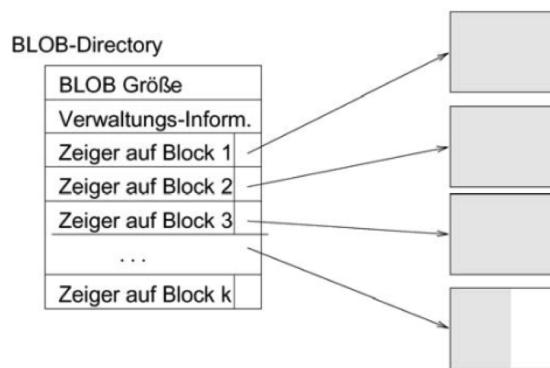


Abbildung 44: BLOB-Speicherung - Lösung 2

- Vorteil: schneller Zugriff auf Teilebereiche des BLOBs
- Nachteil: festgelegte, begrenzte Maximalgröße des BLOBs
(Gigabyte-BLOBs; 8-Byte Adressierung; Seitengröße 1KB \Rightarrow 8MB für ein BLOB-Dictionary)
- effizienter: B-Baum zur Speicherung von BLOBs

4.2.5 TID-Konzept (Adressierung)

- *Tupel-Identifier*(TID) ist eine Datensatz-Adresse bestehend aus Seitennummer und Offset
- Offset verweist innerhalb der Seite bei einem Offsetwert von i auf den i -ten Eintrag in einer Liste von *Tupel-Zeigern*, die am Anfang der Seite stehen.
- Jeder Tupel-Zeiger enthält Offsetwert
- Verschiebung auf der Seite: sämtliche Verweise von außen bleiben unverändert
- Verschiebung auf eine andere Seite: statt altem Datensatz neuer Tupel-Zeiger
- diese zweistufige Referenz aus Effizienzgründen nicht wünschenswert: Reorganisation in regelmäßigen Abständen.

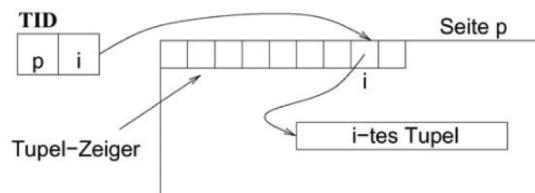


Abbildung 45: TID-Konzept: einstufige Referenz

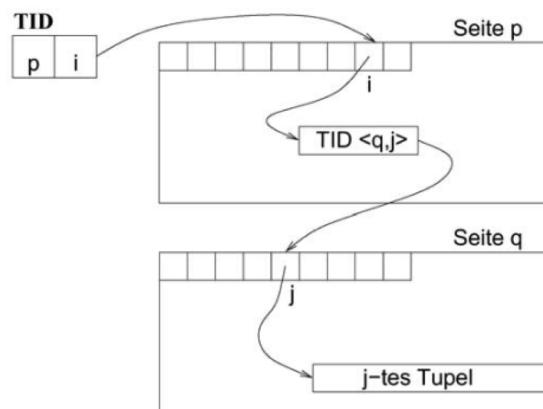


Abbildung 46: TID-Konzept: zweistufige Referenz

4.3 Pufferverwaltung in Detail

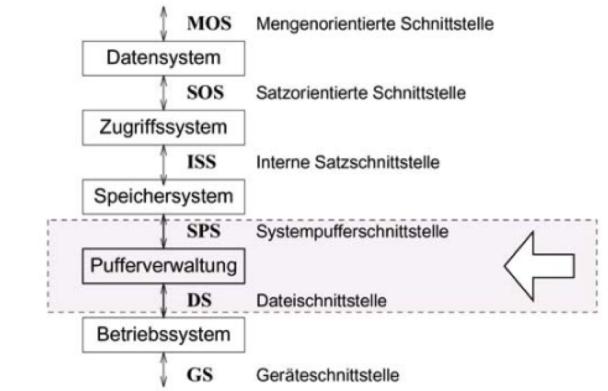


Abbildung 47: Einordnung in Architektur

Der *Puffer* ist ein ausgezeichneter Bereich des Hauptspeichers. Es ist in *Pufferrahmen* gegliedert, jeder Rahmen kann Seite der Platte aufnehmen.

4.3.1 Aufgaben der Pufferverwaltung

- muss an geforderte Seite im Puffer suchen \Rightarrow effiziente *Suchverfahren*
- parallele Datenbanktransaktionen: geschickte *Speicherzuteilung* im Puffer
- Puffer gefüllt \rightarrow adäquate *Seitenersetzungstrategien*
- spezielle Anwendung der Pufferverwaltung: *Schattenspeicherkonzept*

4.3.1.1 Suchverfahren

Direkte Suche: ohne Hilfsmittel linear suchen

Indirekte Suche:

- *unsortierte und sortierte Tabelle*: Alle Seiten im Puffer vermerkt
- *verkettete Liste*: schnelleres sortiertes Einfügen möglich
- *Hash-Tabelle*: bei geschickter gewählter Hash-Funktion günstiger Such- und Änderungsaufwand

4.3.1.2 Speicherzuteilung

bei mehreren parallel anstehenden Transaktionen gibt es verschiedene Strategien:

Lokale Strategien : Jeder Transaktion bestimmte disjunkte Pufferteile verfügbar machen
(Größe statisch vor Ablauf oder dynamisch zur Programmlaufzeit entscheiden)

Globale Strategien : Zugriffsverhalten aller Transaktionen insgesamt bestimmt Speicherzuteilung (gemeinsam von mehreren Transaktionen referenzierte Seiten können so besser berücksichtigt werden)

Seitentypbezogene Strategien : Partition des Puffers: Pufferrahmen für Datenseiten, Zugriffspfadseiten, Data-Dictionary-Seiten, ...

4.3.1.3 Seitenersetzungsstrategien

- Speichersystem fordert E_2 an, die nicht im Puffer vorhanden ist.
- Sämtliche Pufferrahmen sind belegt
- Vor dem Laden von E_2 Pufferrahmen freimachen
- nach den unten beschriebenen Strategien Seite aussuchen
- Ist Seite in der Zwischenzeit im Puffer verändert worden, so wird sie nun auf die Platte *zurückgeschrieben*
- Ist Seite seit Einlagerung in den Puffer nur gelesen worden, so kann sie überschrieben werden (*verdrängt*)

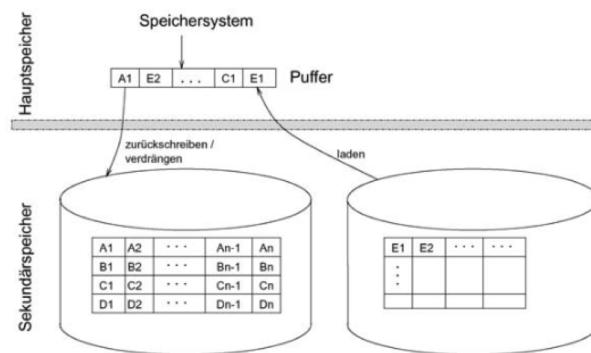


Abbildung 48: Seitenersetzung

Seitenersetzungsverfahren

Demand-paging-Verfahren : genau eine Seite im Puffer durch angeforderte Seite ersetzen

Prepaging-Verfahren : neben der angeforderten Seite auch weitere Seiten in den Puffer einlesen, die eventuell in der Zukunft benötigt werden (z.B. bei BLOBs sinnvoll)

optimale Strategie : Welche Seiten hat maximale Distanz zu ihrem nächsten Gebrauch?
(nicht realisierbar, zukünftiges Referenzverhalten nicht vorhersehbar)

~~> Realisierbare Verfahren besitzen keine Kenntnis über das zukünftige Referenzverhalten

- *Zufallsstrategie*: jeder Seite gleiche Wiederbenutzungswahrscheinlichkeit zuordnen
- Gute, realisierbare Verfahren sollen vergangenes Referenzverhalten auf Seiten nutzen, um Erwartungswerte für Wiederbenutzung schätzen zu können
 - besser als Zufallsstrategie
 - Annäherung an optimale Strategie

Merkmale gängiger Strategien

- *Alter* einer Seite im Puffer:
 - Alter einer Seite nach Einlagerung (die globale Strategie (G))
 - Alter einer Seite nach dem letzten Referenzzeitpunkt (die Strategie des jüngsten Verhaltens (J))
 - Alter einer Seite wird nicht berücksichtigt (-)
- *Anzahl* der Referenzen auf Seite im Puffer:
 - Anzahl aller Referenzen auf einer Seite (die globale Strategie (G))
 - Anzahl nur der letzten Referenz auf einer Seite (die Strategie des jüngsten Verhaltens (J))
 - Anzahl der Referenz wird nicht berücksichtigt (-)

Verfahren	Prinzip	Alter	Anzahl
FIFO	älteste Seite ersetzt	G	–
LFU (least frequently used)	Seite mit geringster Häufigkeit ersetzen	–	G
LRU (least recently used)	Seite ersetzen, die am längsten nicht referenziert wurde (System R)	J	J
DGCLOCK (dyn. generalized clock)	Protokollierung der Ersetzungshäufigkeiten wichtiger Seiten	G	JG
LRD (least reference density)	Ersetzung der Seite mit geringster Referenzdichte	JG	G

Abbildung 49: Klassifikation gängiger Strategien

Mangelnde Eignung des BS-Puffers Natürlicher Verbund von Relationen A und B (zugehörige Folge von Seiten: A_i bzw. B_j) \rightsquigarrow^b Implementierung: *Nested-Loop*

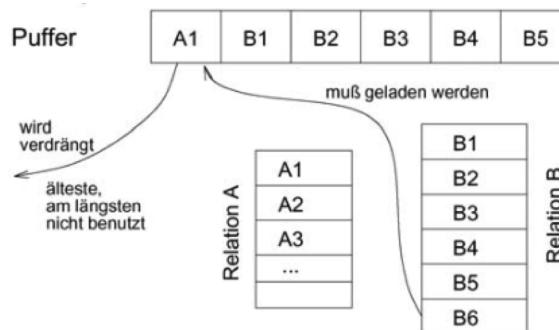


Abbildung 50: Beispiel: Seitenersetzungsverfahren

- FIFO: A_1 verdrängt, da älteste Seite im Puffer
- LRU: A_1 verdrängt, da diese Seite nur im ersten Schritt beim Auslesen des Vergleichstupels benötigt wurde

Problem

- im nächsten Schritt wird A_1 wieder benötigt
- weiteres „Aufschaukeln“: um A_1 laden zu können, muss B_1 entfernt werden (im nächsten Schritt benötigt) usw.

Seitenersetzungsstrategien von Datenbanksystemen

- Fixieren von Seiten
- Freigeben von Seiten
- Zurückschreiben einer Seite (z.B. am Transaktionsende!)

4.3.1.4 Schattenspeicherkonzept

- Modifikation des Pufferkonzepts
- Muss Seite im Verlauf einer Transaktion auf Platte zurückgeschrieben werden: nicht auf die Orginalseite sondern auf neue Seite
- statt einer Hilfsstruktur zwei parallele Hilfsstrukturen
- Hilfsstrukturen: *virtuelle Seitentabellen*

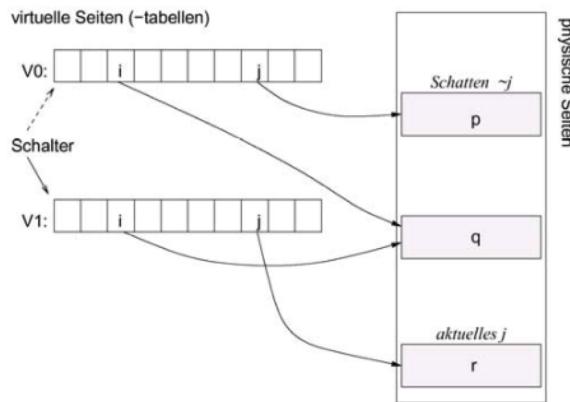


Abbildung 51: Schattenspeicherkonzept

- bei Transaktionsabbruch, nur Zugriff auf Ursprungsversion V_0 notwendig, um den alten Zustand der Seite vor Beginn der Transaktion wiederherzustellen.
- Umschalten zwischen den beiden Versionen der virutellen Seitentabellen durch einen „Schalter“.
- Transaktion erfolgreich beendet $\Rightarrow V_1$ wird Orginalversion und V_0 verweist dann auf die neuen Schattenseiten.
- Nachteil: ehemals zusammenhängende Seitenbereiche einer Relation werden nach Änderungsoperationen auf der Datenbank über den Sekundärspeicher „verstreut“.

5 Dateiorganisation und Zugriffsstrukturen

Einordnung

- *Speichersystem* fordert über Systempufferschnittstelle Seiten an
- interpretiert diese als *interne Datensätze*
- interne Realisierung der logischen Datensätze mit Hilfe von Zeigern, speziellen Indexeinträgen und weiteren Hilfsstrukturen
- *Zugriffssystem* absrahiert von konkreter Realisierung

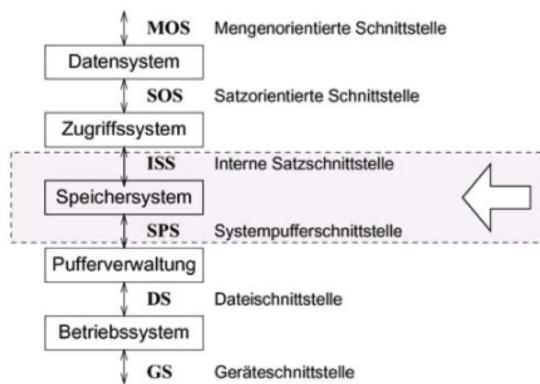


Abbildung 52: Einordnung in Architektur

5.1 Klassifikation der Speicherungstechniken

Kriterien für *Zugriffsstrukturen* oder *Zugriffsverfahren*:

- organisiert interne Relation selbst (Dateiorganisationsform) oder zusätzliche Zugriffsmöglichkeit auf bestehende interne Relation (Zugriffspfad)
- Art der Zuordnung von gegebenen Attributwerten zu Datensatz-Adressen
- Arten von Anfragen, die durch Dateiorganisationsformen und Zugriffspfade effizient unterstützt werden können

5.1.1 Primär- vs. Sekundärschlüssel

Unterscheidungsmerkmale: Art des / der unterstützten Attribute(s)

Primärschlüssel

- Duplikatfreiheit
- identifizierende Attributmenge
- Verbundoperationen oft über Primärschlüssel

Sekundärschlüssel (beliebige andere Attributmengen)

- meist keine Schlüsseleigenschaften
- meist kein identifizierendes Merkmal
- Unterstützung bestimmter Anfragen (etwa Selektion über Sekundärschlüsseln)

Weitere Unterscheidungsmerkmale

- Normalfall: Primärschlüssel über *Primärindex* oder *Dateiorganisationsform* unterstützt, kann *geclusteter Index* sein.
- Normalfall: Sekundärschlüssel über *Sekundärindex* oder *Zugriffspfad* unterstützt, Index ist nicht geclustert.
- *dünnbesetzter Index* eignet sich nur für Primärschlüssel.
- *dichtbesetzter Index* muss bei Sekundärschlüssel eingesetzt werden.

5.1.2 Primär- vs. Sekundärindex

Primärindex Zugriffspfad auf interne Relation, welcher Dateiorganisation oder interne Relation ausnutzen kann.

- interne Relation \Rightarrow Primärindex kann Sortierung aussnutzen \Rightarrow *geclusteter Index* oder *dünnbesetzter Index* möglich
- im Normalfall über Primärschlüssel aber auch über Sekundärindex denkbar

Sekundärindex jeder weitere Zugriffspfad auf interne Relation, welcher Dateiorganisation der internen Relation aussnutzen kann

- *nicht-geclusteter Index* oder *dichtbesetzter Index*
- pro interne Relation ein Primärindex, mehrere Sekundärindexe
- einige RDBS: Sekundärindexe als Zugriffspfad \Rightarrow kein Index nutzt Art der Speicherung der internen Relation aus

5.1.3 Dateiorganistaion vs. Zugriffspfad

Dateiorganisationsform: Form der Speicherung einer internen Relation

- unsortierte Speicherung von internen Tupeln: *Heap-Organisation*
- sortierte Speicherung von internen Tupeln: *sequenzielle Organisation*
- gestreute Speicherung von internen Tupeln: *Hash-Organisation*
- Speicherung von interenen Tupel in mehrdimensionalen Räumen: *mehrdimensionale Dateiorganisation*
- Üblich: Sortierung oder Hash-Funktion über Primärschlüssel
- sortierte Speicherung plus zusätzlicher Primärindex über Sortierattribut: *indexsequenzielle Organisationsform*

Zugriffspfad: Über grundlegende Dateiorganisationsform hinausgehende Zugriffsstruktur, etwa Indexdatei

- Einträge ($K, K \uparrow$)
- K der Wert eines Primär- oder Sekundärschlüssels
- $K \uparrow$ Datensatz oder Verweis auf Datensatz
- K : Suchschlüssel, genauer Zugriffsattribute und Zugriffsattributwerte

Eintrag $K \uparrow$:

- $K \uparrow$ ist *Datensatz* selbst: Zugriffspfad wird Dateiorganisationsform
- $K \uparrow$ is *Adresse eines internen Tupels* : Primärschlüssel; Sekundärschlüssel mit $(K, K \uparrow_1), \dots, (K, K \uparrow_n)$ für denselben Zugriffsattributwert K
- $K \uparrow$ ist *Liste von Tupeladressen*: Sekundärschlüssel; nachteilig ist variable Länge der Indexeinträge

Tupeladressen: TIDs, nur Seitenadressen, . . .

Indexdatei kann selbst in Dateiorganisationsform gespeichert werden und mit Zugriffspfaden versehen werden

5.1.4 Dünn- vs. Dichtbesetzter Index

- *dünnbesetzter Index*: nicht für jeden Zugriffsattributwert K ein Eintrag in Indexdatei
 - interne Relation sortiert nach Zugriffsattributen: im Index steht ein Eintrag pro Seite \Rightarrow Index verweist mit $(K_1, K_1 \uparrow)$ auf *Seitenanführer*, nächster Indexeintrag $(K_2, K_2 \uparrow)$
 - Datensatz mit Zugriffsattributwert $K_?$ mit $K_1 \leq K_? < K_2$ ist auf Seite von $K_1 \uparrow$ zu finden

- *indexsequenzielle Datei*: sortierte Datei mit dünnbesetztem Index als Primärindex
- *dichtbesetzter Index*: für jeden Datensatz der internen Relation ein Eintrag in Indexdatei
- Primärindex kann dichtbesetzter Index sein, wenn Dateiorganisationenform Heap-Datei, aber auch bei Sortierung (*geclusterter Index*)

5.1.5 (Nicht-)Geclusterter Index

geclusteter Index In der gleichen Form sortiert wie interne Relation

- Bsp.: interne Relation *Student* Matrikelnummer sortiert \Rightarrow Indexdatei über dem Attribut *Matrikelnummer* üblicherweise geclustert.

nicht geclusteter Index Anders organisiert als interne Relation

- Bsp.: über *Studiengang* ein Sekundärindex, Datei selbst nach Matrikelnummern sortiert.
- Primärindex kann dünnbesetzt und geclustert sein
- Jeder dünnbesetzte Index ist auch ein geclusteter Index, aber nicht umgekehrt
- Sekundärindex kann nur dichtbesetzter, nicht-geclusteter Index sein (auch: invertierte Datei)

5.1.6 Schlüsselzugriff vs. -Transformation

Schlüsselzugriff Zuordnung von Primär- oder Sekundärschlüsselwerten zu Adressen in Hilfsstruktur wie Indexdatei

- Bsp.: indexsequenzielle Organisation, B-Baum, KdB-Baum, ...

Schlüsseltransformation Berechnet Tupeladresse aufgrund einer Formel aus Primär- und Sekundärschlüsselwerten (statt Indexeinträgen nur Berechnungsvorschrift gespeichert)

- Bsp.: Hash-Verfahren

5.1.7 Ein-Attribut vs. Mehr-Attribut-Index

Ein-Attribut-Index (*non-composite index*) Zugriffspfad über einem einzigen Zugriffsattribut

- immer *eindimensionale Zugriffsstruktur*: Zugriffsattibutwerte definieren lineare Ordnung in eindimensionalem Raum.

Mehr-Attribut-Index (*composite index*) Zugriffspfad über mehrere Attribute

- Vorteil: bei *exact-match* nur ein Indexzugriff (weniger Seitenzugriffe)
- Ausführungsart bestimmt, ob neben *exact-match* auch noch *partial-match* effizient unterstützt wird (eindimensional oder mehrdimensional)
- Ist *eindimensionale Zugriffsstruktur* oder *mehrdimensionale*
 - *eindimensionaler Fall*: Kombination der verschiedenen Zugriffsattributwerte konkateniert, wird als ein einziges Zugriffattribut betrachtet (wieder lineare Ordnung in eindimensionalem Raum)
- Bsp.: Attribut *Vorname* und *PLZ* unterstützen entweder zwei Ein-Attribut-Indexe oder Zwei-Attribut-Indexe über beiden Attributen.
- *eindimensionaler Fall*: (*Vorname*, *PLZ*) unterstützt kein *partial-match* nach *PLZ*.
- *mehrdimensionaler Fall*: Menge der Zugriffsattributwerte spannt mehrdimensionalen Raum auf ⇒ bei *partial-match* bestimmt horizontale oder vertikale Grenze im Raum die Treffermenge. Somit sind auch Anfragen nach *PLZ* mit *partial-match* möglich.

Vorname	PLZ	Adresse
Andreas	18209	•
Christa	69121	•
Gunter	39106	•

Abbildung 53: Beispiel: ein- vs. mehrdimensionale Struktur

5.1.8 Statische vs. Dynamische Struktur

statische Zugriffsstruktur Optimal nur bei bestimmter (fester) Anzahl von verwalteten Datensätzen.

- Bsp. 1: Adresstransformation für Personalausweisnummer p von Personen mit $p \bmod 5$
 5 Seiten, Seitengröße 1KB, durchschnittliche Satzlänge 200 Bytes, Gleichverteilung der Personalausweisnummern
 ⇒ für 25 Personen optimal, für 10.000 Personen nicht mehr ausreichend.

- Bsp. 2: Telefonbücher einer Firma

Indexstruktur dreistufig: Bereichsverzeichnis, in Telefonbuch Index auf Ort, innerhalb eines Ortes Nutzer nach Nachnamen sortiert. Die Firma hat 30 Kunden insgesamt \Rightarrow nur eine Seite nötig, keine drei Indexstufen nötig. Die Firma wird Weltkonzern und hat 3 Milliarden Kunden: nun ist die dreistufige Indexstruktur nicht mehr ausreichend, mindestens eine vierte Stufe für das Länderverzeichnis ist nötig.

dynamische Zugriffsstruktur Unabhängig von der Anzahl der Datensätze optimal

- dynamische Adresstransformationsverfahren verändern dynamisch Bildbereich der Transformation
- dynamische Indexverfahren verändern dynamisch Anzahl der Indexstufen \Rightarrow in DBS üblich

5.1.9 Klassifikation

- Beispiel für dynamisches und eindimensionales Verfahren: *B-Baum* (beliebtester Zugriffspfad in relationalen Datenbanksystemen)
- wird meistens mit `create index` angelegt
- nur *exact-match*, kein *partial-match*
- Beispiel für statisches und eindimensionales Verfahren: *klassisches Hashverfahren*

5.1.10 Anforderungen an Speichertechniken

- dynamisches Verhalten
- Effizienz bei Einzelzugriff (Schlüsselsuche beim Primärindex)
- Effizienz beim Mehrfachzugriff (Schlüsselsuche beim Sekundärindex)
- Ausnutzung für sequenziellen Durchlauf (Sortierung, geclusterte Indexe)
- Clustering
- Anfragetypen: *exact-match*, *partial-match*, *range queries* (Bereichsanfragen)

5.2 Statische Verfahren

- Heap, indexsequenziell, indiziert-nichtsequenziell
- oft grundlegende Speichertechnik in RDBS

- *direkten Organisationsformen*: keine Hilfstruktur, keine Adressberechnung (Heap, sequenziell)
- statische Indexverfahren für Primärindex und Sekundärindex

5.2.1 Heap-Organisation

- völlig unsortiert speichern
- physische Reihenfolge der Datensätze ist zeitliche Reihenfolge der Aufnahme von Datensätzen

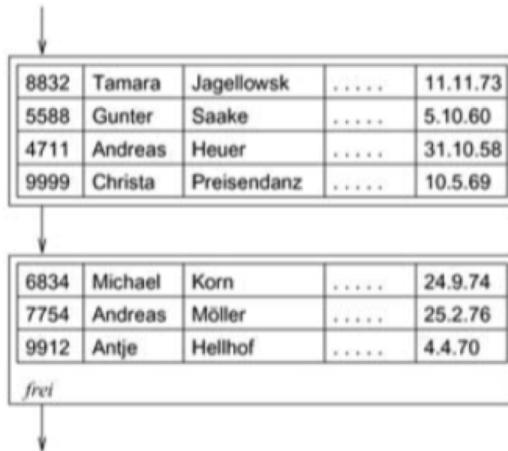


Abbildung 54: Beispiel: Heap-Organisation

Operationen - Heap-Organisation

- **insert**: Zugriff auf letzte Seite der Datei. Genügend freier Platz \Rightarrow Satz anhängen.
Sonst nächste freie Seite holen
- **delete**: **lookup**, dann Löschbit auf 0 gesetzt
- **lookup**: sequenzielles Durchsuchen der Gesamtdatei, maximaler Aufwand (Heap-Datei meist zusammen mit Sekundärindex eingesetzt; oder für sehr kleine Relationen)
- Komplexitäten: Neuaufnahme von Daten $\mathcal{O}(1)$, Suchen $\mathcal{O}(n)$

5.2.2 Seqenzielle Speicherung

- sortiertes Speichern der Datensätze

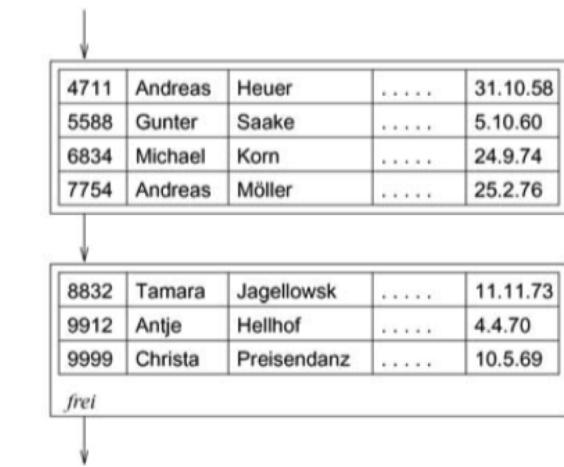


Abbildung 55: Beispiel: sequenzielle Speicherung

Operationen

- **insert:** Seite suchen, Datensatz einsortieren ⇒ beim Anlegen oder sequenziellen Füllen einer Datei jede Seite nur bis zu gewissem Grad (etwa 66 Prozent) füllen
- **delete:** Aufwand bleibt
- Folgende Dateiorganisationsformen:
 - schnellere **lookup**
 - mehr Platzbedarf (durch Hilfsstrukturen wie Indexdateien)
 - mehr Zeitbedarf bei **insert** und **delete**
- klassische Indexform: indexsequenzielle Dateiorganisation

5.2.2.1 Indexsequenzielle Dateiorganisation

- Kombination aus sequenzieller Hauptdatei und Indexdatei: *indexsequenzielle Dateiorganisationsform*
- Indexdatei kann geclusterter, dünnbesetzter Index sein
- mindestens zweistufiger Baum
 - Blattebene ist *Hauptdatei* (Datensätze)
 - jede andere Stufe ist *Indexdatei*

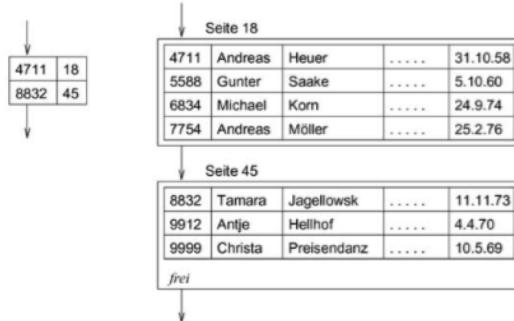


Abbildung 56: Beispiel: Indexsequenziell

Aufbau der Indexdatei - Indexsequenziell

- Datensätze in Indexdatei: (Primärschlüssel, Seitennummer) zu jeder Seite der Hauptdatei genau ein Index-Datensatz in Indexdatei
- Problem: “Wurzel” des Baumes bei einem einstufigen Index nicht nur eine Seite

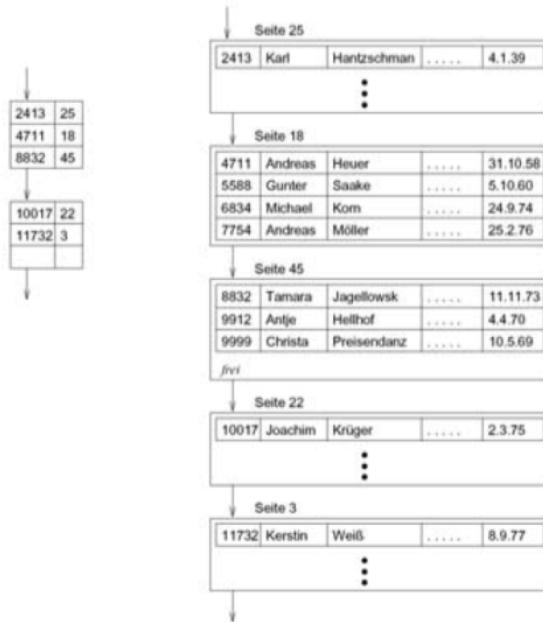


Abbildung 57: Aufbau der Indexdatei- indexsequ

Lookup bei indexsequenziellen Dateien lookup-Operation sucht Datensatz zum Zugriffsattributwert ω . Indexdatei sequenziell durchlaufen, dabei (v_1, s) im Index gesucht mit $v_1 \leq \omega$:

- (v_1, s) ist letzter Satz der Indexdatei, dann kann Datensatz zu ω höchstens auf dieser Seite gespeichert sein (wenn er existiert)
- nächster Satz (v_2, \bar{s}) im Index hat $v_2 > \omega$, also muss Datensatz zu ω , wenn vorhanden, auf Seite s gespeichert sein

(v_1, s) überdeckt Zugriffsattributwert ω

Insert bei indexsequenziellen Dateien

- **insert:** zunächst **lookup** Seite 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 die alte und neue Seite verteilen; neue Seite Indexeintrag anlegen
- Alternativ neuen Datensatz auf Überlaufseite zur gefundenen Seite

Delete bei indexsequenziellen Dateien

- **delete:** zunächst lookup Seiten finden
- Satz auf Seite löschen (Löschbit auf 0)
- erster Satz auf Seite: Index anpassen
- Falls Seite nach Löschen leer: Index anpassen, Seite an Freispeicherverwaltung zurück

Probleme bei indexsequenziellen Dateien

- *stark wachsende Dateien:* Zahl der linear verketten Indexseiten wächst; automatische Anpassung der Stufenanzahl nicht vorgesehen
- *stark strumpfende Dateien:* nur zögernde Verringerung der Index- und Hauptseiten
- *unausgeglichene Seiten* in der Hauptdatei (unnötig hoher Speicherplatzbedarf, zu lange Zugriffszeiten)

5.2.2.2 Mehrstufiger Index

- Optional: Indexdatei wieder indexsequenziell verwalten
- Idealerweise: Index höchster Stufe nur noch eine Seite

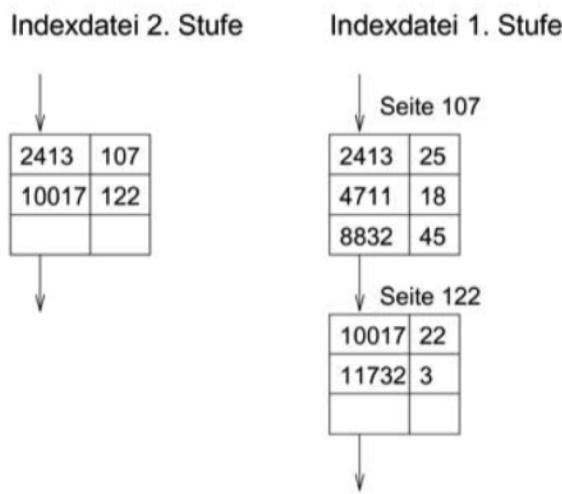


Abbildung 58: Beispiel: sequenzielle Speicherung

5.2.2.3 Indizierter nicht-sequenzieller Zugriffspfad

- zur Unterstützung von Sekundärschlüsseln
- mehrere Zugriffspfade dieser Form pro Datei möglich
- einstufig oder mehrstufig: höhere Indexstufen wieder indexsequenziell organisiert

Aufbau der Indexdatei - nicht-sequenzieller Zugriffspfad

- Sekundärindex, dichbesetzter und nicht-geclusteter Index
- Zu jedem Satz der Hauptdatei Satz (ω, s) in der Indexdatei
- ω Sekundärschlüsselwert, s zugeordnete Seite
 - entweder für ω mehrere Sätze in die Indexdatei aufnehmen
 - oder für ein ω Liste von Adressen in der Hauptdatei angeben

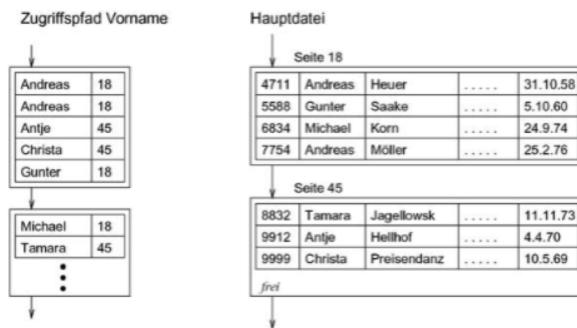


Abbildung 59: Aufbau der Indexdatei - 1

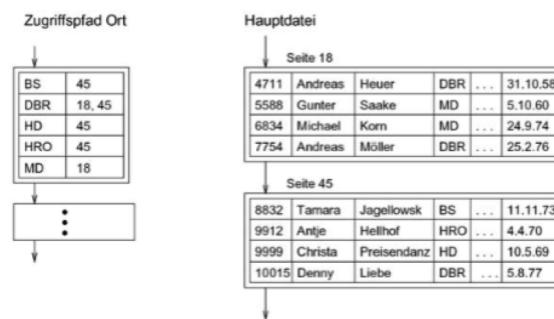


Abbildung 60: Aufbau der Indexdatei - 2

Operationen - nicht-sequenzieller Zugriffspfad

- **loopup:** ω kann mehrfach auftreten, Überdeckungstechniken nicht benötigt
- **insert:** Anpassen der Indexdateien
- **delete:** Indexeintrag entfernen

5.3 Baumverfahren

5.3.1 B-Bäume

- Ausgangspunkt: ausgeglichener, balancierter Suchbaum
- *Ausgeglichen* oder *balanciert*: alle Pfade von der Wurzel des Baumes zu den Blättern gleich lang
- Hauptspeicher-Implementierungsstruktur: binäre Suchbäume
- Datenbankbereich: Knoten der Suchbäume zugeschnitten auf Seitenstruktur des Datenbanksystems

- mehrere Zugriffsattributwerte auf einer Seite
- *Mehrweg-Bäume*

5.3.1.1 Prinzip des B-Baums

- *B-Baum* von Bayer (B für balanciert, breit, buschig, Bayer, **NICHT:** binär)
- dynamischer, balancierter Indexbaum, bei dem jeder Indexeintrag auf eine Seite der Hauptdatei zeigt

Mehrwegebaumbau völlig ausgeglichen, wenn

1. alle Wege von Wurzel zu Blättern gleich lang
2. jeder Knoten gleich viele Indexeinträge

vollständiges Ausgleichen zu teuer, deshalb *B-Baum-Kriterium*:

Jede Seite außer der Wurzelseite enthält zwischen m und 2m Daten

5.3.1.2 Eigenschaften

n Datensätze in der Hauptdatei \Rightarrow in $\log_m(n)$ Seitenzugriffen von der Wurzel zum Blatt

- Durch Balancierungskriterium wird Eigenschaft nahe an der vollständigen Ausgeglichenheit erreicht (1. Kriterium vollständig erfüllt, 2. näherungsweise)
- Kriterium garantiert 50 Prozent Speicherplatzauslastung
- einfache, schnelle Algorithmen zum Suchen, Einfügen und Löschen von Datensätzen (Komplexität von $\mathcal{O}(\log_m(n))$)
- B-Baum als Primär- und Sekundärindex geeignet
- Datensätze direkt in die Indexseite \Rightarrow Dateiorganisationsform
- Verweist man aus Indexseite auf Datensätze in der Hauptseite \Rightarrow Sekundärindex

5.3.1.3 Definition

Ordnung eines B-Baumes ist minimale Anzahl der Einträge auf den Indexseiten außer der Wurzelseite.

Bsp.: B-Baum der Ordnung 8 fasst auf jeder inneren Indexseite zwischen 8 und 16 Einträgen

Ein *Indexbaum* ist ein B-Baum der Ordnung m , wenn er die folgenden Eigenschaften erfüllt:

1. Jede Seite enthält höchstens $2m$ Elemente

2. Jede Seite, außer der Wurzelseite, enthält mindestens m Elemente
 3. Jede Seite ist entweder eine Blattseite ohne Nachfolger oder hat $i + 1$ Nachfolger, falls i die Anzahl ihrer Elemente ist.
 4. Alle Blattseiten liegen auf der gleichen Stufe

Suchen im B-Baum

- **lookup** wie in statischen Indexverfahren
 - Startet auf Wurzelseite Eintrag in B-Baum ermitteln, der den gesuchten Zugriffssatztributwert ω überdeckt \Rightarrow Zeiger verfolgen, Seite nächster Stufe laden

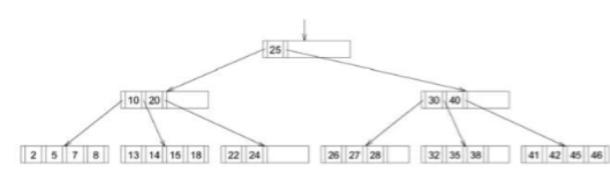


Abbildung 61: Suchen in B-Baum (38,20,6)

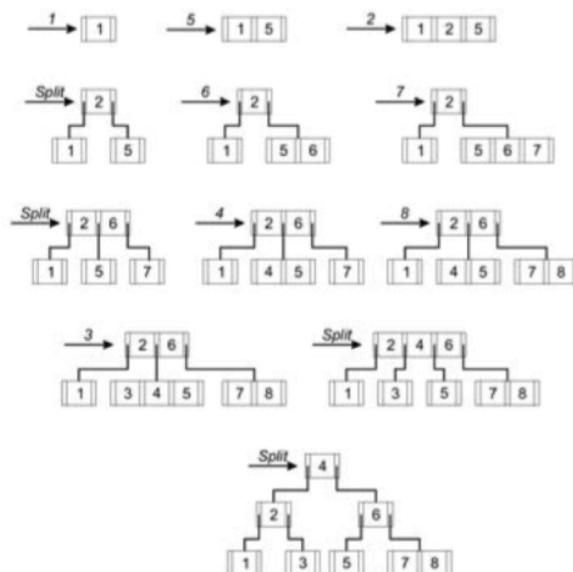


Abbildung 62: Einfügen in B-Baum

Einfügen in B-Baum Einfügen eines Wertes ω

- mit **lookup** entsprechende Blattseite suchen

- passende Seite $n < 2m$ Elemente, ω einsortieren
- passende Seite $n = 2m$ Elemente, neue Seite erzeugen
 - ersten m Wert auf Orgnialseite
 - letzten m Wert auf neue Seite
 - mittleres Element auf entsprechende Indexseite nach oben
- eventuell dieser Prozess rekursiv bis zur Wurzel

Löschen in B-Baum Bei weniger als m Elementen auf Seite: Unterlauf Löschen eines Wertes ω :

- mit **lookup** entsprechende Seite suchen
- ω auf Blattseite gespeichert \Rightarrow Wert löschen, evtl. Unterlaufbehandlung
- ω nicht auf Blattseite gespeichert \Rightarrow Wert löschen, durch lexographisch nächstkleineres Element von einer Blattseite ersetzen, evtl. Unterlaufbehandlung

Unterlaufbehandlung

- Ausgleichen mit der benachbarten Seite (benachbarte Seite n Elemente mit $n > m$)
- oder Zusammenlegen zweier Seiten zu einer (Nachbarseite $n = m$ Elemente), das “mittlere” Element von Indexseite darüber dazu, auf Indexseite evtl. Unterlauf behandeln

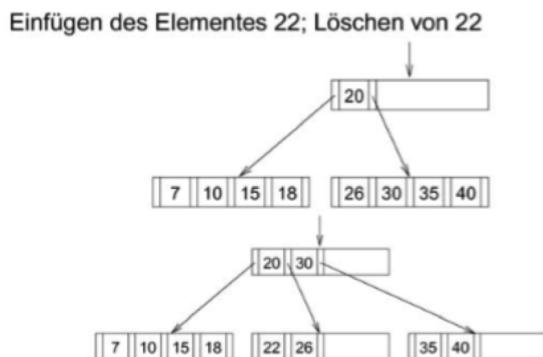


Abbildung 63: Einfügen und Löschen

5.3.1.4 Komplexität der Operationen

- Aufwand beim Einfügen, Suchen und Löschen immer $\mathcal{O}(\log_m(n))$ Operationen
- entspricht genau der “Höhe” des Baumes
- Konkret: Seiten der Größe 4KB, Zugriffsattributwert 32 Bytes, 8-Byte-Zeiger: zwischen 50 und 100 Indexeinträge pro Seite: Ordnung dieses Baumes 50
- 1.000.000 Datensätze: $\log_5 0(1.000.000) = 4$ Seitenzugriffe im schlechtesten Fall
- Wurzelseiten jedes B-Baumes normalerweise im Puffer: drei Seitenzugriffe

5.4 Hash-Verfahren

- Schlüsseltransformation und Überlaufbehandlung
- DB-Technik: Bildbereich entspricht Seiten-Adressraum
- Dynamik: dynamische Hash-Funktionen oder Re-Hashen

5.4.1 Grundprinzipien

- Basis-Hash-Funktion: $h(k) = k \bmod m$
- m möglichst Primzahl
- Überlaufbehandlung
 - Überlaufseiten als verkettete Liste
 - lineares Sondieren
 - quadratisches Sondieren
 - doppeltes Hashen

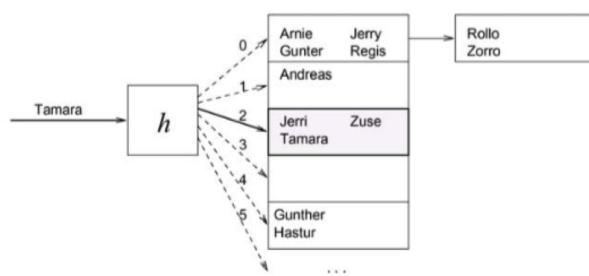


Abbildung 64: Hash-Verfahren in Datenbanken

5.4.2 Operationen und Zeitkomplexität

- **lookup, modify, insert, delete**
- **lookup** benötigt maximal $1 + \#B(h(\omega))$ Seitenzugriffe
- $\#B(h(\omega))$ Anzahl der Seiten (inklusive der Überlaufseiten) des Buckets für Hash-Wert $h(\omega)$
- Unter Schranke 2 (Zugriff auf Hash-Verzeichnis plus Zugriff auf erste Seite)

5.4.3 Statisches Hashen: Probleme

- mangelnde Dynamik
- Vergrößerung des Bildbereichs erfordert komplettes Neu-Hashen
- Wahl der Hash-Funktion entscheidend!
- Bsp.: Hash-Index aus 100 Buckets, Studenten über 6-stellige *Matrikelnummer* (wird fortlaufend vergeben) hashen:
 - ersten beiden Stellen: Datensätze auf wenigen Seiten quasi sequenziell abgespeichert
 - letzten beiden Stellen: verteilen die Datensätze gleichmäßig auf alle Seiten
- Sortiertes Ausgeben einer Relation schlecht

5.4.4 Lineares Hashen

Folge von Hash-Funktionen, die wie folgt charakterisiert sind:

- $h_i : \text{dom}(\text{Primärschlüssen}) \rightarrow 0, \dots, 2^i \times N - 1$ ist eine Folge von Hash-Funktionen mit $i \in 0, 1, 2, \dots$ und N als Anfangsgröße des Hash-Verzeichnisses
- Wert von i wird auch als *Level* der Hash-Funktion bezeichnet
 $\text{dom}(\text{Primärschlüssel})$ wir im folgenden als $\text{dom}(\text{Prim})$ abgekürzt.
- Für diese Hash-Funktionen gelten die folgenden Bedingungen:
 - $h_{i+1}(\omega) = h_i(\omega)$ für etwa die Hälfte aller $\omega \in \text{dom}(\text{Prim})$
 - $h_{i+1}(\omega) = h_i(\omega) + 2^i \times N$ für die andere Hälfte

Bedingungen sind zum Beispiel erfüllt, wenn $h_i(\omega)$ als $\omega \bmod (2^i \times N)$ gewählt wird

5.4.4.1 Prinzip

Für ein ω höchstens zwei Hash-Funktionen zuständig, deren Level nur um 1 differiert, Entscheidung zwischen diesen beiden durch *Split-Zeiger*

- sp Split-Zeiger (gibt an, welche Seite als nächstes geteilt wird)
- lv Level (gibt an, welche Hash-Funktionen benutzt werden)

Aus Split-Zeiger und Level lässt sich Gesamtanzahl A der belegten Seiten wie folgt berechnen:

$$Anz = 2^{lv} + sp \quad (10)$$

Beide Werte werden am Anfang mit 0 initialisiert.

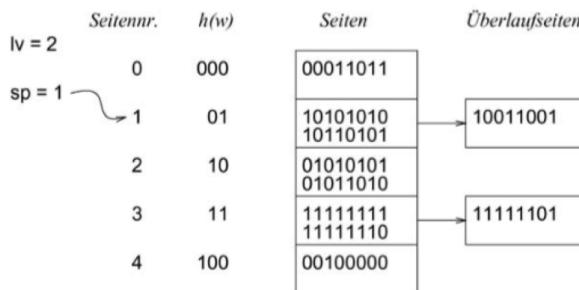


Abbildung 65: Lineares Hashen

Lookup

- zuerst Hash-Wert mit der “kleineren” Hash-Funktion bestimmen
- liegt dieser unter dem Wert des Split-Zeigers \Rightarrow größere Hash-Funktion verwenden.

```

 $s := h_{lv}(w);$ 
if  $s < sp$ 
then  $s := h_{lv+1}(w);$ 

```

Abbildung 66: Lookup

Splitten einer Seite

1. Die Sätze der Seite (Bucket), auf die sp zeigt, werden mittels h_{lv-1} neu verteilt (ca. die Hälfte der Sätze wird auf Seite (Bucket) unter Hash-Nummern $2^{lv} * N + sp$ verschoben)

2. Der Split-Zeiger wird weitergesetzt: $sp := sp + 1$;
3. Nach Abarbeiten eines Levels wird wieder bei Seite 0 begonnen; der Level wird um 1 erhöht

```
if  $sp = 2^{lv} * N$  then
  begin
     $lv := lv + 1;$ 
     $sp := 0$ 
  end;
```

Abbildung 67: Splitten einer Seite

	Seitennr.	$h(w)$	Seiten	Überlaufseiten
lv = 2	0	000	00011011	
sp = 2	1	001	10011001	
	2	10	01010101 01011010	
	3	11	11111111 11111110	11111101
	4	100	00100000	
	5	101	10101010 10110101	

Nach Split von Seite 1

Abbildung 68: Lineares Hashen

Problem des Linearen Hashen .

$lv = 3$	$Seitennr.$	$h(w)$	$Seiten$	$Überlaufseiten$
sp = 0	0	000	00011011	
	1	001	10011001	
	2	010	01010101 01011010	
	3	011		
	4	100	00100000	
	5	101	10101010 10110101	
	6	110		
	7	111	11111111 11111110	11111101

Abbildung 69: Problem lineares Hashen

Verbesserungen

- Erweiterbares Hashen
- Spiral-Hashen

5.5 Mehrdimensionale Speichertechniken

- bisher: eindimensional (keine *partial-match* Anfragen nur lineare Ordnung)
- jetzt: mehrdimensional (auch *partial-match* Anfragen, Positionierung im mehrdimensionalen Datenraum)
- k Dimensionen = k Attribute können gleichberechtigt unterstützt werden

5.5.1 Mehrdimensionale Baumverfahren

KdB-Baum ist B⁺-Baum, bei dem Indexseite als binäre Bäume mit Zugriffsattributen, Zugriffsattributwerten und Zeigern realisiert werden. Varianten von k -dimensionalen Indexbäumen:

- *kd-Baum*: für Hauptspeicheralgorithmen entwickelte mehrdimensionale Grundstruktur (binärer Baum)
- *KDB-Baum*: Kombination kd-Baum und B-Baum (k -dimensionaler Indexbaum bekommt höheren Verzweigungsgrad)
- *KdB-Baum*: Verbesserung des KDB-Baums (den schauen wir uns an!)

5.5.1.1 Kdb-Bäume

- Kann Primär- und mehrere Sekundärschlüssel gleichzeitig unterstützen
- macht als Dateiorganisationsform zusätzliche Sekundärindexe überflüssig

Definition Idee: auf jeder Indexseite einen Teilbaum darstellen, der nach mehreren Attributen hintereinander verzweigt.

- *KdB-Baum vom Typ(b, t)* besteht aus
 - inneren Knoten (Bereichsseiten) die ein *kd-Baum* mit maximal b internen Knoten enthalten.
 - Blätter (Satzseiten) die bis zu t Tupel der gespeicherten Relation speichern können.
- Bereichsseiten: *kd-Baum* enthalten mit *Schnittelelementen* und zwei Zeigern
 - Schnittelelement enthält *Zugriffsattribut* und *Zugriffsattributwerte*; rechter Zeiger: größere Zugriffsattributwerte

Struktur

- Bereichsseiten
 - Anzahl der Schnitt- und Adressenelemente der Seite
 - Zeiger auf Wurzel des in der Seite enthaltenen kd-Baumes
 - *Schnitt- und Adressenelemente*
- Schnittelelement
 - Zugriffsattribut
 - Zugriffsattributwert
 - zwei Zeiger auf Nachfolgerknoten des kd-Baumes dieser Seite (können Schnitt oder Adressenelemente sein)
- Adressenelemente
 - Adresse eines Nachfolgers der Bereichsseite im KdB-Baum (Bereichs- oder Satzseite)

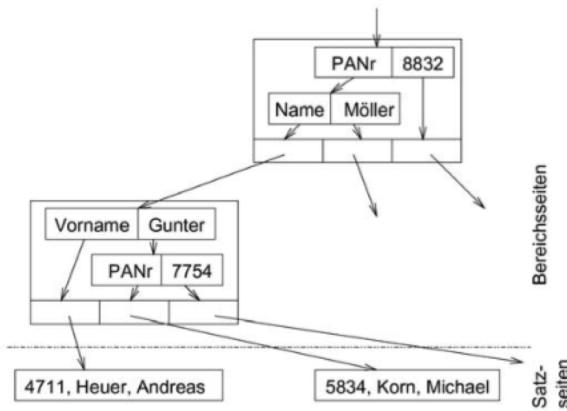


Abbildung 70: KdB-Baum

Operationen - Kdb-Bäume

- Komplexität **lookup**, **insert** und **delete** bei *exact-match* logn
- bei *partial-match* besser als $\mathcal{O}(n)$
- bei t von k Attributen in der Anfrage spezifiziert: Zugriffskomplexität von $\mathcal{O}(n^{1-t/k})$

5.5.2 Trennattribute

- Reihenfolge der Trennattribute
 - entweder zyklish festgelegt
 - oder Selektivität einbeziehen: Zugriffsattribut mit hoher Selektivität sollte früher und häufiger als Schnittelement eingesetzt werden
- Trennattributwert: Aufgrund von Informationen über Verteilung von Attributwerten eine geeignete “Mitte” eines aufzutrennenden Attributwertbereichs ermitteln

5.5.3 Brickwall-Darstellung

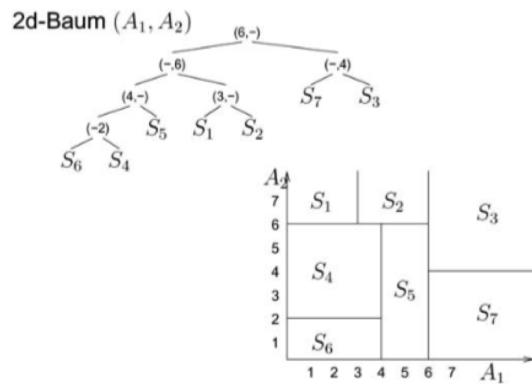


Abbildung 71: 2d-Baum

5.5.4 Mehrdimensionales Hashing

- Idee: Bit Interleaving
- abwechselnd von verschiedenen Zugriffsattributwerden die Bits der Adressen berechnen

	*0*0	*0*1	*1*0	*1*1
0*0*	0000	0001	0100	0101
0*1*	0010	0011	0110	0111
1*0*	1000	1001	1100	1101
1*1*	1010	1011	1110	1111

Abbildung 72: Beispiel: 2 Dimensionen

5.5.4.1 Idee

- MDH baut auf linearem Hashing auf

- Hash-Werte sind Bit-Folgen, von denen jeweils ein Anfangsstück als aktueller Hash-Wert dient
- je ein Bit-String pro beteiligtem Attribut berechnen
- Anfangsstücke nun nach dem Prinzip des Bit-Interleaving zyklisch abarbeiten
- Hash-Wert reihum aus den Bits der Einzelwerte zusammensetzen

5.5.4.2 Veranschaulichung

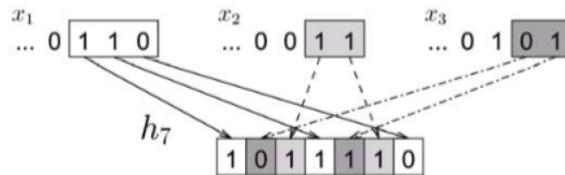


Abbildung 73: Veranschaulichung

- verdeutlicht Komposition der Hash-Funktion h_i für drei Dimensionen und den Wert $i = 7$
- graphisch unterlegte Teile der Bit-Strings entsprechen den Werten $h_{7_1}(x_1)$, $h_{7_2}(x_2)$ und $h_{7_3}(x_3)$
- beim Schritt auf $i = 8$ würde ein weiteres Bit von x_2 (genauer: von $h_{8_2}(x_2)$) verwendet

5.5.4.3 Komplexität

- Exact-Match-Anfragen: $\mathcal{O}(1)$
- Partial-Match-Anfragen, bei t von k Attributen festgelegt, Aufwand $\mathcal{O}(n^{1-t/k})$
- ergibt sich aus der Zahl der Seiten, wenn bestimmte Bits “unknown”
- Spezialfälle: $\mathcal{O}(1)$ für $t = k$, $\mathcal{O}(n)$ für $t = 0$

5.5.5 Grid-Files

- bekannteste und von der Technik her die attraktivste mehrdimensionale Dateiorganisationsform
- eigene Kategorie: Elemente der Schlüsseltransformation wie bei Hash-Verfahren und Indexdatei wie bei Baumverfahren kombiniert
- mehrdimensionaler Raum sehr “gleichmäßig” aufgeteilt (im Gegensatz zu Brickwall)

5.5.5.1 Zielsetzung

- Prinzip der 2 Plattenzugriffe: Jeder Datensatz soll bei einer *exact-match*-Anfrage in 2 Zugriffen erreichbar sein
- Zerlegung des Datenraums in Quader: n -dimensionale Quader bilden die Suchregion im Grid-File
- Prinzip der Nachbarschaftserhaltung: Ähnliche Objekte sollten auf der gleichen Seite gespeichert werden
- Symmetrische Behandlung aller Raum-Dimensionen: *partial-match*-Anfragen ermöglicht
- Dynamische Anpassung der Grid-Struktur beim Einfügen und Löschen

Prinzip der zwei Plattenzugriffe Beim *exact-match* Beim *exact-match*

1. gesuchtes k -Tupel auf Intervalle der *Skalen* abbilden; als Kombination der ermittelten Intervalle werden Indexwerte erreicht; Skalen im Hauptspeicher \Rightarrow noch kein Plattenzugriff
2. über errechnete Indexwerte Zugriff auf das *Grid-Directory*; dort sind die Adressen der Datensatz-Seiten gespeichert; **erster Plattenzugriff**
3. der Datensatz-Zugriff: **zweiter Plattenzugriff**

5.5.5.2 Aufbau eines Grid-Files

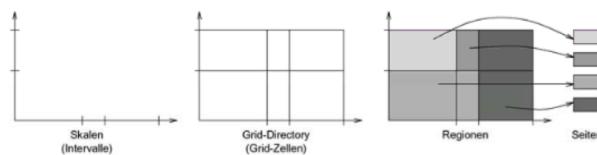


Abbildung 74: Aufbau eines Grid-Files

- *Grid*: k eindimensionale Felder (Skalen), jede Skala repräsentiert Attribut
- *Skalen* bestehen aus Partitionen der zugeordneten Wertebereiche in *Intervalle*
- *Grid-Directory* besteht aus Grid-Zellen, die den Datenraum in Quader zerlegen
- *Grid-Zellen* bilden eine Grid-Region, der genau eine Datensatz-Seite zugeordnet wird
- *Grid-Region*: k -dimensionales, konvexe Gebiet. (Regionen sind paarweise disjunkt)

5.5.5.3 Operationen - Grid-Files

Zu Anfang: Zelle = Region = eine Datensatz-Seite

- *Seitenüberlauf*: Seite wird geteilt. Falls die zur Seite gehörende Grid-Region aus nur einer Grid-Zelle besteht, muss ein Intervall auf einer Skala in zwei Intervalle unterteilt werden. Besteht die Region aus mehreren Zellen, so werden diese Zellen in einzelne Regionen zerlegt.
- *Seitenunterlauf*: Zwei Regionen zu einer zusammenfassen, falls das Ergebnis eine neue konvexe Region ergibt.

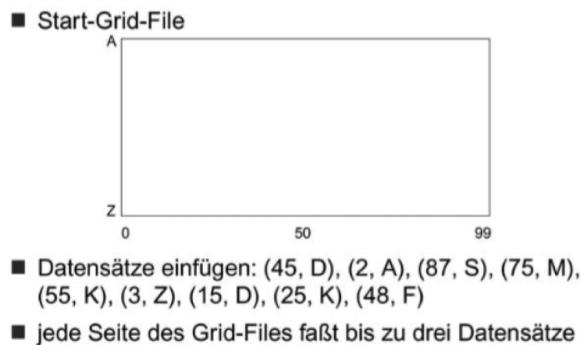


Abbildung 75: Beispiel: Grid-Files

5.5.5.4 Buddy-System

- Die im gleichen Schritt entstandenen Zellen können zu Regionen zusammengefasst werden; Keine andere Zusammenfassung von Zellen ist im Buddy-System erlaubt
- Unflexibel beim Löschen: nur Zusammenfassung von Regionen erlaubt, die vorher als Zwillinge entstanden waren

6 Basisalgorithmen für Datenbankoperationen

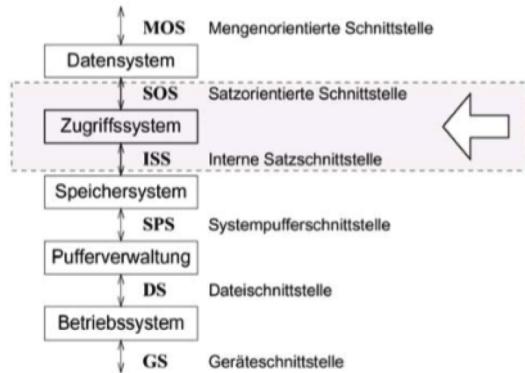


Abbildung 76: Einordnung in Architektur

6.1 Datenbankparameterter

- Komplexitätsbetrachtung ($\mathcal{O}(n^2)$)
- Aufwandsabschätzungen (konkret)
- Datenbankparameter als Grundlage
- müssen im Katalog des DBS gespeichert werden

- n_r : Anzahl Tupel in Relation r
- b_r : Anzahl von Blöcken (Seiten), die Tupel aus r beinhalten
- s_r : (mittlere) Größe von Tupeln aus r (s für size)
- f_r : Blockungsfaktor (Tupel aus r pro Block)

$$f_r = \frac{bs}{s_r},$$

mit bs Blockgröße

- Tupel einer Relation kompakt in Blöcken:

$$b_r = \left\lceil \frac{n_r}{f_r} \right\rceil$$

Abbildung 77: Datenbankparameter

- $V(A, r)$: Anzahl verschiedener Werte für das Attribut A in der Relation r (V für *values*): $V(A, r) = |\pi_A(r)|$
 - A Primärschlüssel: $V(A, r) = n_r$
 - $SC(A, r)$: Selektionskardinalität (*selection cardinality*); durchschnittliche Anzahl von Ergebnistupeln bei $\sigma_{A=x}(r)$ für $x \in \pi_A(r)$
 - Schlüsselattribut A : $SC(A, r) = 1$
 - Allgemein:
- $$SC(A, r) = \frac{n_r}{V(A, r)}$$

Abbildung 78: Datenbankparameter

Weiterhin: Verzweigungsgrad bei B-Baum-Indexen, Höhe des Baums, Anzahl von Blätterknoten

6.2 Grundannahmen

- Indexe B⁺-Bäume
- dominierender Kostenfaktor: Blockzugriff
- Zugriff auf Hintergrundspeicher auch für Zwischenrelationen
- Zwischenrelationen zunächst für jede Gruppenoperation
- Zwischenrelationen hoffentlich zu großen Teil im Puffer
- einige Operationen (Mengenoperationen) auf Adressmengen (TID-Listen)

6.3 Grundalgorithmen

6.3.1 Hauptspeicheralgorithmen

Wichtig für den Durchsatz des Gesamtsystems, da sie sehr oft eingesetzt werden

- *Tupelvergleich*
(Duplikate erkennen, Sortierordnung angeben, ...)
iterativ durch Vergleich der Einzelattribute, Attribut mit großer Selektivität zuerst
- *TID-Zugriff*
TID innerhalb des Hauptspeichers: übliche Vorgehensweise bei der Auflösung indirekter Adressen

6.3.2 Zugriffe auf Datensätze

- *Relationen*: interner Identifikator **RelID**
- *Indexe*: interner Identifikator **IndexID**
 - *Primärindex*, etwa $I(\text{Personen}(PANr))$
bei $A = a$ wird maximal ein Tupel pro Zugriff
 - *Sekundärindex*, etwa $I(\text{Ausleihe}(PANr))$
Bsp.: $PANr = 4711$ liefert i.a. mehrere Tupel

Indexzugriffe: Ergebnis TID-Listen

- **fetch-tupel**: Direktzugriff auf Tupel mittels TID-Wertes
holt Tupel in *Tupel-Puffer*
 $\text{fetch-tupel}(\text{RelID}, \text{TID}) \rightarrow \text{Tupel-Puffer}$
- **fetch-TID**: TID zu (Primärschlüssel-)Attributwert
bestimmen
 $\text{fetch-TID}(\text{IndexID}, \text{Attributwert}) \rightarrow \text{TID}$
- weiterhin auf Relationen und Indexen: *Scans*

Abbildung 79: Zugriffe auf Datensätze

```
select *  
from KUNDE  
where KName ='Meier'  
  
■ Gleichheitsanfrage über einen Schlüssel  
■ put: hier Anzeige des Ergebnisses  
  
aktuellerTID :=  
    fetch-TID(KUNDE-KName-Index, 'Meyer');  
aktuellerPuffer:=  
    fetch-tupel(KUNDE-RelationID, aktuellerTID);  
put(aktuellerPuffer);
```

Abbildung 80: Beispiel in SQL

6.3.3 Externes Sortieren

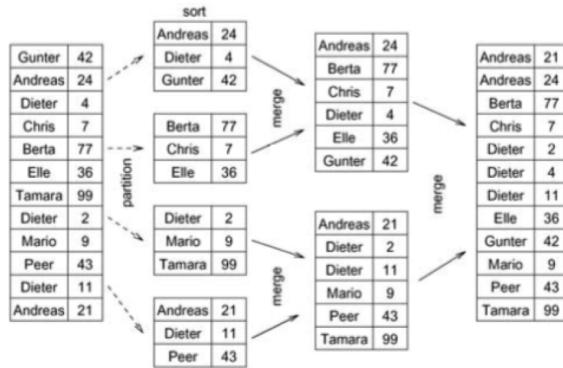


Abbildung 81: Beispiel externes Sortieren

Externes Sortieren durch Mischen; Komplexität $\mathcal{O}(n \log n)$ Vertausoperationen
 \Rightarrow Dabei geht es nicht um den Sortieralgorithmus, sondern um die Möglichkeit einen großen Datenbereich von der Platte sortiert zu laden.
 Wird verwendet, wenn große Datenbereiche nicht in den Hauptspeicher passen um diese zu sorteiren.

1. Partitionierungsphase

- Aufteilung der Orginaltupel / tabellen in Stücke die man im Hauptspeicher sortiert:

$$\frac{b_r}{mem} Stcken \quad (11)$$

- Seperate Sortierung in Zwischenrelationen

2. Mischphase

- Zwei oder mehrere Zwischenrelationen werden blockweise eingeladen und gemischt
- bei Mischung von jeweils 2 Zwischenergebnissen 2^n Partitionen in n Mischläufen

$$\frac{b_r}{mem} Partitionen \rightsquigarrow \log_2\left(\frac{b_r}{mem}\right) Mischlufen \quad (12)$$

- pro Teilrelation ein Block
 \rightarrow dann können jeweils $mem - 1$ in einem Durchgang gemischt werden $\Rightarrow \log_{mem-1}\left(\frac{b_r}{mem}\right)$ Mischläufe

Gesamtaufwand

- Jeder Block wird zum Schreiben einmal gelesen und einmal geschrieben
- Jeder Mischlauf liest und schreibt jeden Block einmal

$$cost_{sort} = 2b_r(1 + \log_{mem-1}\left(\frac{b_r}{mem}\right)) \quad (13)$$

- Im Beispiel: 1 Tupel pro Block, $b_r = 12$, $mem = 3$, pro Mischlauf $mem - 1 = 2$ Partitionen mischen.

$$\Rightarrow \log_2\left(\frac{12}{3}\right) = 2 \text{ Mischlufe} \quad (14)$$

6.4 Unäre Operationen

Scan durchläuft Tupel einer Relation

Relationen-Scan (*full table scan*) Durchläuft alle Tupel einer Relation in beliebiger Reihenfolge

Aufwand: b_r

Index-Scan Nutzt Index zum Auslesen der Tupel in Sortierreihenfolge

Aufwand: Anzahl der Tupel plus Höhe des Indexes

Vergleich

- Relationen-Scan besser durch Ausnutzung der Blockung
- Index-Scan besser, falls wenige Daten benötigt, aber schlechter beim Auslesen vieler Tupel

6.4.1 Operationen auf Scans

- Relationen-Scan öffnen
open-rel-scan(RelationenID) → ScanID
 liefert ScanID zurück, die bei folgenden Operationen zur Identifikation genutzt wird
- Index-Scan initialisieren
open-index-scan(IndexID, Min, Max) → ScanID
 liefert ScanID zurück; Min und Max bestimmen Bereich einer Bereichsanfrage
- **next-TID** liefert nächsten TID; Scan-Cursor weitersetzen
- **end-of-scan** liefert **true**, falls kein TID mehr im Scan abzuarbeiten
- **close-scan** schließt Scan

Abbildung 82: Operationen auf Scans

```
select *
from Personen
where Nachname between 'Heuer' and
'Jagellowsk'
```

Abbildung 83: Beispiel Scans

```
aktuellerScanID := open-rel-scan(Personen-RelationID);
aktuellerTID := next-TID(aktuellerScanID);
while not end-of-scan(aktuellerScanID) do
begin
    aktuellerPuffer := 
        fetch-tupel(Personen-RelationID,aktuellerTID);
    if aktuellerPuffer.Nachname >= 'Heuer'
        and aktuellerPuffer.Nachname <= 'Jagellowsk'
    then put (aktuellerPuffer);
    endif;
    aktuellerTID := next-TID(aktuellerScanID);
end;
close (aktuellerScanID);
```

Abbildung 84: Beispiel Relationen-Scan

```
aktuellerScanID :=
    open-index-scan(Personen-Nachname-IndexID,
    'Heuer','Jagellowsk');
aktuellerTID := next-TID(aktuellerScanID);
while not end-of-scan(aktuellerScanID) do
begin
    aktuellerPuffer :=
        fetch-tupel(Personen-RelationID,aktuellerTID)
    put(aktuellerPuffer);
    aktuellerTID := next-TID(aktuellerScanID);
end;
close (aktuellerScanID);
```

Abbildung 85: Beispiel Index-Scan

6.4.2 Selektion

- *exakte Suche, Basis selektion*, komplexe zusammgesetzte Selektionskriterien
- Zusammengesetztes Prädikat φ aus automaren Prädikaten (exakte Suche, Bereichsanfrage) mit **and**, **or**, **not**

Tupelweise Vorgehen:

- Gegeben $\sigma_\varphi(r)$
- Relationen-Scan: für alle $t \in r$ auswerten $\varphi(t)$
- Aufwand $\mathcal{O}(n_r)$, genauer b_r

6.4.3 Projektion

- Relationsalgebra: mit Duplikateliminierung
- SQL: keine Duplikateliminierung, wenn nicht mit **distinct** gefordert (modifizierter Scan)
- mit Duplikateliminierung:
 - sortierte Ausgabe eines Indexes hilft bei der Duplikateliminierung
 - Projektion auf indexierte Attribute ohne Zugriff auf gespeicherte Tupel

- **Projektion** $\pi_X(r)$:
 1. r nach X sortieren
 2. $t \in r$ werden in das Ergebnis aufgenommen, für die $t(X) \neq \text{previous}(t(X))$ gilt
- **Zeitaufwand:** $O(n_r \log n_r)$
- Falls r schon sortiert nach X : $O(n_r)$
- **Schlüssel** $K \subseteq X$: $O(n_r)$

Abbildung 86: Projektion

6.5 Binäre Operationen

Binäre Operationen meist auf Basis von tupelweisem Vergleich der einzelnen Tupelmengen

- *Nested-Loops-Techniken* oder *Schleifeniteration*
 - für jedes Tupel einer äußeren Relation s wird die innere Relation r komplett durchlaufen
 - Aufwand: $\mathcal{O}(n_s * n_r)$
- *Merge-Techniken* oder *Mischmethode*
 - r und s (sortiert) schrittweise in der vorgegebenen Tupelreihenfolge durchlaufen
 - Aufwand: $\mathcal{O}(n_s + n_r)$
 - Falls Sortierung nicht vorhanden: *Sort-Merge-Technik*
 - Aufwand: $n_r \log n_r$ und / oder $n_s \log n_s$
- *Hash-Methoden*
 - kleinere oder beiden Relationen in Hash-Tabelle
 - Tupel der zweiten Relation finden ihren Vergleichspartner mittels Hash-Funktion
 - idealerweise Aufwand: $\mathcal{O}(n_s + n_r)$

6.5.1 Vereinigung

Vereinigung durch Einfügen

- Variante der Nested-Loop-Methoden
- Kopie einer der beiden Relationen r_2 unter dem Namen \bar{r}_2 anlegen, dann Tupel $t_1 \in r_1$ in \bar{r}_2 einfügen
(Zeitaufwand abhängig von Organisationsform der Kopie)

Spezialtechniken für die Vereinigung

- r und s verkettet
- Projektion auf alle Attribute der verketteten Relation

Zeitaufwand: $\mathcal{O}((n_r + n_s) \times \log(n_r + n_s))$ (wie Projektion)

Vereinigung durch Merge-Techniken (**merge-union**)

1. r und s sortieren, falls nicht bereits sortiert
 2. r und s mischen
 - $t_r \in r$ kleiner als $t_s \in s$: t_r in das Ergebnis, nächstes $t_r \in r$ lesen
 - $t_r \in r$ größer als $t_s \in s$: t_s in das Ergebnis, nächstes $t_s \in s$ lesen
 - $t_s = t_r$: t_r in das Ergebnis, nächste $t_r \in r$ bzw. $t_s \in s$ lesen
- Zeitaufwand: $O(n_r \times \log n_r + n_s \times \log n_s)$ mit Sortierung, $O(n_r + n_s)$ ohne Sortierung

Abbildung 87: Vereinigung

6.5.1.1 Join-Berechnung

Varianten

- Nested-Loops-Verbund
- Block-Nested-Loops-Verbund
- Merge-Join
- Hash-Verbund
- ...

Nested-Loop-Verbund Doppelte Schleife iteriert über alle $t_1 \in r$ und alle $t_2 \in s$ bei einer Operation $r \bowtie_\varphi s$:

$r \bowtie_\varphi s$:

```
for each  $t_r \in r$  do
begin
    for each  $t_s \in s$  do
    begin
        if  $\varphi(t_r, t_s)$  then put( $t_r \cdot t_s$ )
    end
end
```

Abbildung 88: Nested-Loop-Verbund

```
R1ScanID := open-rel-scan(R1ID);
R1TID := next-TID(R1ScanID);
while not end-of-scan(R1ScanID) do
begin
    R1Puffer := fetch-tupel(R1ID,R1TID);
    R2ScanID := open-rel-scan(R2ID);
    R2TID := next-TID(R2ScanID);
    while not end-of-scan(R2ScanID) do
    begin
        /* Scan über innere Relation */
    end;
    close (R2ScanID);
    R1TID := next-TID(R1ScanID);
end;
close (R1ScanID);
```

Abbildung 89: Nested-Loop-Verbund mit Scan - 1

```
/* Scan über innere Relation */
R2Puffer := fetch-tupel(R2ID,R2TID);
if R1Puffer.X = R2Puffer.Y
then insert into ERG
    (R1.Puffer.A1, ..., R1.Puffer.An, R1.Puffer.X,
     R2.Puffer.B1, ..., R1.Puffer.Bm);
endif;
R2TID := next-TID(R2ScanID);
```

Abbildung 90: Nested-Loop-Verbund mit Scan - 2

Verbesserung: Nested-Loops-Verbund verbindet alle $t_1 \in r$ mit Ergebnis von $\omega_{X=t_1(X)}(s)$ (gut bei Index auf X in r_2)

Block-Nested-Loop-Verbund Statt über Tupel über Blöcke iterieren:

```

for each Block  $B_r$  of  $r$  do
begin
    for each Block  $B_s$  of  $s$  do
        begin
            for each Tupel  $t_r \in B_r$  do
                begin
                    for each Tupel  $t_s \in B_s$  do
                        begin
                            if  $\varphi(t_r, t_s)$  then put( $t_r \cdot t_s$ ) endif
                        end
                    end
                end
            end
        end
    Aufwand:  $b_r * b_s$ 

```

Abbildung 91: Block-Nested-Loop-Verbund

Merge-Techniken

$X := R \cap S$; falls nicht bereits sortiert, zuerst Sortierung von r und s nach X

1. $t_r(X) < t_s(X)$, nächstes $t_r \in r$ lesen
2. $t_r(X) > t_s(X)$, nächstes $t_s \in s$ lesen
3. $t_r(X) = t_s(X)$, t_r mit t_s und allen Nachfolgern von t_s , die auf X mit t_s gleich, verbinden
4. beim ersten $t'_s \in s$ mit $t'_s(X) \neq t_s(X)$ beginnend mit ursprünglichem t_s mit den Nachfolgern t'_r von t_r wiederholen, solange $t_r(X) = t'_r(X)$ gilt

Abbildung 92: Merge-Techniken

Merge-Join mit Scan

- Verbund-Attribute auf beiden Relationen Schlüsseleigenschaften
- **max(X)** und **min(X)**: minmaler bzw. maximaler gespeicherter Wert für X

```

R1ScanID := open-index-scan(R1XIndexID,
    min(X), max(X));
R1TID := next-TID(R1ScanID);
R1Puffer := fetch-tupel(R1ID,R1TID);
R2ScanID := open-index-scan(R2YIndexID,
    min(Y), max(Y));
R2TID := next-TID(R2ScanID);
R2Puffer := fetch-tupel(R2ID,R2TID);
while not end-of-scan(R1ScanID)
    and not end-of-scan(R2ScanID) do
begin
    .../* merge */
end;
close (R1ScanID);
close (R2ScanID);

```

Abbildung 93: Merge-Join mit Scan - 1

```

/* merge */
if R1Puffer.X < R2Puffer.Y
then R1TID := next-TID(R1ScanID);
    R1Puffer := fetch-tupel(R1ID,R1TID);
else if R1Puffer.X > R2Puffer.y
then R2TID := next-TID(R2ScanID);
    R2Puffer := fetch-tupel(R2ID,R2TID);
else insert into ERG
    (R1.Puffer.A1, ..., R1.Puffer.An, R1.Puffer.X,
     R2.Puffer.B1, ..., R1.Puffer.Bm);
    R1TID := next-TID(R1ScanID);
    R1Puffer := fetch-tupel(R1ID,R1TID);
    R2TID := next-TID(R2ScanID);
    R2Puffer := fetch-tupel(R2ID,R2TID);
endif;
endif;

```

Abbildung 94: Merge-Join mit Scan - 2

Verbund durch Hashing Idee:

- Ausnutzung des verfügbaren Hauptspeichers zur Minimierung der Externspeicherzugriffe
- Finden der Verbundpartner durch Hashing
- Anfragen der Form $r \bowtie_{r.A=s.B} s$

Classic Hashing

- Vorbereitung: kleinere Relation wird r
- Ablauf:
 1. Tupel von r mittels Scan in Hauptspeicher lesen und mittels Hashfunktion $h(r.A)$ in Hashtabelle H einordnen

2. wenn H (oder r vollständig gelesen): Scan über S und mit $h(s.B)$ Verbundpartner suchen
3. Falls Scan über r nicht abgeschlossen:
 H neu aufbauen und erneuten Scan über S durchführen
 - Auwand: $\mathcal{O}(b_r + p * b_s)$ mit p Anzahl der Scans über s

Simpler Hash-Verbund

- Verhindere mehrfaches Lesen der Tupel aus s
- Wähle jeweils Bildbereich $h(A) = h(B)$ für r bzw. s so, dass alle Tupel in den Hauptspeicher passen
- Tupel aus r mit passenden Hash-Werten (= “im aktiven Bereich”) werden per Hash-Tabelle in Bucket eingetragen, Rest geht in Temp-Bereich für r
- Tupel aus s mit passenden Hash-Werten werden mit Tupel aus korrespondierenden Buckets verbunden, Rest geht in Temp-Bereich für s
- Wiederholen mit nächstem Ausschnitt der Temp-Bereiche

Partitionierung mittels Hashfunktion

“Hash-Partitionierter Verbund”

- Tupel aus r und s über X in gemeinsame Datei mit k Blöcken (Buckets) “gehasht”
- Tupel in gleichen Buckets durch Verbundalgorithmus verbinden

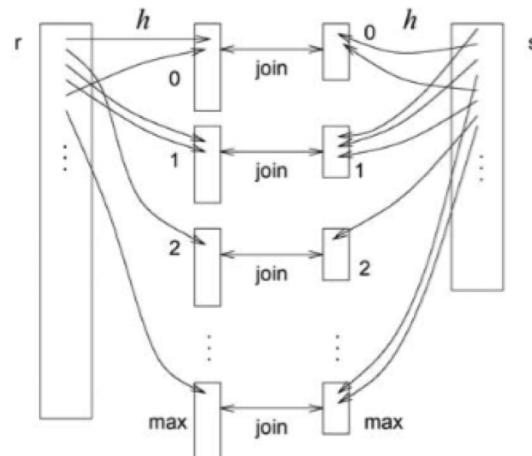


Abbildung 95: Beispiel: Partitionierung

```
for each  $t_r$  in  $r$  do
begin
     $i := h(t_r(X))$ ;
     $H_i^r := H_i^r \cup t_r(X)$ ;
end;
for each  $t_s$  in  $s$  do
begin
     $i := h(t_s(X))$ ;
     $H_i^s := H_i^s \cup t_s(X)$ ;
end;
for each  $k$  in  $0 \dots \max$  do
 $H_k^r \bowtie H_k^s$ ;
```

Abbildung 96: Partitionierung mittels Hashfunktion

7 Anfrageoptimierung

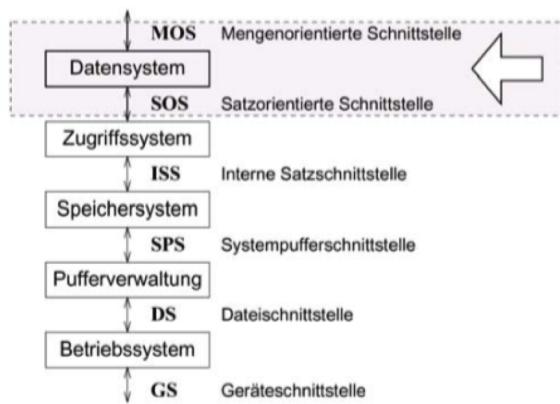


Abbildung 97: Einordnung in Architektur

7.1 Grundprinzipien

Basissprachen:

- SQL
- Relationskalküle
- hier: Relationenalgebra

Ziel der Optimierung:

- möglichst schnelle Anfragebearbeitung ⇒
- möglichst wenig Seitenzugriffe bei der Anfragebearbeitung ⇒
- möglichst in allen Operationen so wenig wie möglich Seiten (Tupel) berücksichtigen

7.1.1 Teilziele der Optimierung

1. Selektion so früh wie möglich
2. Basisoperationen zusammenfassen ohne Zwischenspeicherung realisieren
3. Redundante Operationen, Idempotenzien oder leere Zwischenoperationen entfernen
4. Zusammenfassen gleicher Teilausdrücke: Wiederverwendung von Zwischenergebnissen

```
KUNDE { KName, Kadr, Kto }
AUFTRAG { KName, Ware, Menge }

select KUNDE.KName, Kto
from KUNDE, AUFTRAG
where KUNDE.KName = AUFTRAG.KName
and Ware = 'Kaffee'

■ Relation KUNDE: 100 Tupel; eine Seite: 5 Tupel
■ Relation AUFTRAG: 10.000 Tupel; eine Seite: 10 Tupel
■ 50 der Aufträge betreffen Kaffee
■ Tupel der Form (KName, Kto): 50 auf eine Seite
■ 3 Zeilen von KUNDE × AUFTRAG auf eine Seite
■ Puffer für jede Relation Größe 1, keine Spannsätze
```

Abbildung 98: Beispiel

7.1.2 Optimierte Auswertung

Ca. 1.140 Seitenzugriffe (Faktor 500 verbessert)

1. $R_1 := \sigma_{\text{Ware}='Kaffee'}(\text{AUFTRAG})$
 - $l : 10.000 / 10 = 1.000$
 - $s : 50 / 10 = 5$
2. $R_2 := \text{KUNDE} \bowtie_{\text{KName}=\text{KName}} R_1$
 - $l : 100 / 5 * 5 = 100$
 - $s : 50 / 3 = 17$
3. $ERG := \pi_{\text{PROJ}}(R_2)$
 - $l : 17$
 - $s : 1$

Abbildung 99: Optimierung Auswertung

7.1.3 Auswertung mit Index

Indexe $I(AUFTRAG(Ware))$ und $I(KUNDE(KName))$

1. $R_1 := \sigma_{\text{Ware}='Kaffee'}(\text{AUFTRAG}) \text{ über } I(\text{AUFTRAG(Ware)})$
 ■ $l : \text{minimal } 5, \text{ maximal } 50; s : 50/10 = 5$
2. $R_2 := \text{sortiere } R_1 \text{ nach KName}$
 ■ $l + s : 5 * \log 5 = 15 \text{ (ca.)}$
3. $R_3 := \text{KUNDE} \bowtie_{\text{KName}=\text{KName}} R_2$
 ■ $l : 100/5 + 5 = 25; s : 50/3 = 17$
4. $ERG := \pi_{\text{PROJ}}(R_3)$
 ■ $l : 17; s : 1$

Abbildung 100: Auswertung mit Index

Maximal ca. 130 und minimal ca. 85 Seitenzugriffe

7.2 Phasen der Anfrageverarbeitung

1. Übersetzung und Sichtenexpansion

- in Anfrageplan arithmetische Ausdrücke vereinfachen
- Unteranfragen auflösen
- Einsetzen der Sichtendefinition

2. Logische oder auch algebraische Optimierung

- Anfrageplan unabhängig von der konkreten Speicherungsform umformen; etwa Hineinziehen von Selektion in andere Operationen

3. Interne Optimierung

- konkrete Speicherungstechniken (Indexe, Cluster) berücksichtigen
- Algorithmen auswählen
- mehrere alternative interne Pläne

4. Kostenbasierte Auswahl

- Statiskinformation (Größe von Tabellen, Selektivität von Attributen) für die Auswahl eines konkreten internen Planes nutzen

5. Code-Erzeugung

- Umwandlung des Zugriffsplans in ausführbaren Code

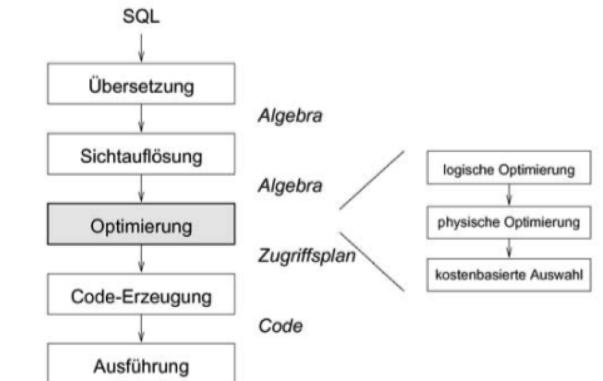


Abbildung 101: Ablauf und Sprachen

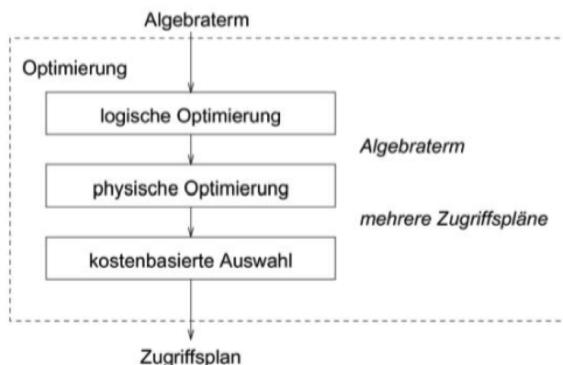


Abbildung 102: Phasen der Optimierung

7.3 Übersetzung in Relationsalgebra

```

select A1, ..., Am
from R1, R2, ..., Rn
where F
Umsetzung in Relationenalgebra:

```

$$\pi_{A_1, \dots, A_m}(\sigma_F(r(R_1) \times r(R_2) \times r(R_3) \times \dots \times r(R_n)))$$

Abbildung 103: Übersetzung in Relationsalgebra

Anfragebaum (folgende Abbildung) verbessern gemäß:

- Erkennen von Verbunden statt Kreuzprodukt
- Auflösung von Unteranfragen (**not exists**-Anfrage in Differenz)

- SQL-Konstrukte, die in der Relationsalgebra kein Gegenstück haben: **group by**, **order by**, Arithmetik, Multimengensemantik

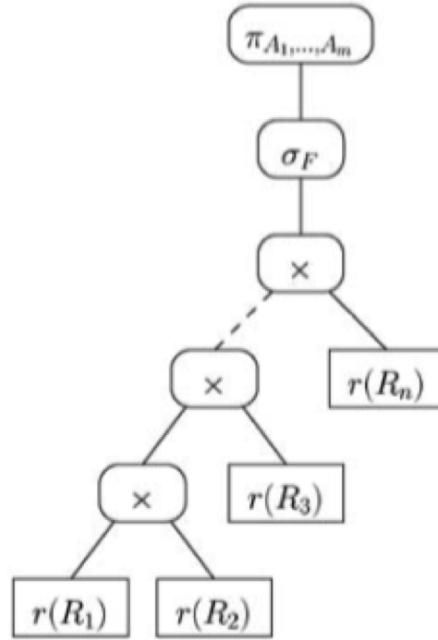


Abbildung 104: Umsetzung des SFW-Blocks

7.3.1 Normalisierung

- Vereinfachung der folgenden Optimierungsschritte durch ein einheitliches (kanonisches) Anfrageformat
- speziell für Selektions- und Verbundbedingungen
 - *konjunktive Normalform* vs. *disjunktive Normalform*
 - konjunktive Normalform (KNF) für einfache Prädikate p_{ij} :

$$(p_{11} \vee p_{12} \vee \dots \vee p_{1n}) \wedge \dots \wedge (p_{m1} \vee p_{m2} \vee \dots \vee p_{mn}) \quad (15)$$

– disjunktive Normalform (DNF):

$$(p_{11} \wedge p_{12} \wedge \dots \wedge p_{1n}) \vee \dots \vee (p_{m1} \wedge p_{m2} \wedge \dots \wedge p_{mn}) \quad (16)$$

– Überführung in KNF / DNF durch Anwendung von Äquivalenzbeziehungen für logische Operationen

■ Äquivalenzbeziehungen

- ◆ $p_1 \wedge p_2 \longleftrightarrow p_2 \wedge p_1$ **und** $p_1 \vee p_2 \longleftrightarrow p_2 \vee p_1$
- ◆ $p_1 \wedge (p_2 \wedge p_3) \longleftrightarrow (p_1 \wedge p_2) \wedge p_3$ **und**
 $p_1 \wedge (p_2 \vee p_3) \longleftrightarrow (p_1 \vee p_2) \vee p_3$
- ◆ $p_1 \wedge (p_2 \vee p_3) \longleftrightarrow (p_1 \wedge p_2) \vee (p_1 \wedge p_3)$ **und**
 $p_1 \vee (p_2 \wedge p_3) \longleftrightarrow (p_1 \vee p_2) \wedge (p_1 \vee p_3)$
- ◆ $\neg(p_1 \wedge p_2) \longleftrightarrow \neg p_1 \vee \neg p_2$ **und**
 $\neg(p_1 \vee p_2) \longleftrightarrow \neg p_1 \wedge \neg p_2$
- ◆ $\neg(\neg p_1) \longleftrightarrow p_1$

Abbildung 105: Normalisierung

■ Anfrage:

```
select * from Projekt P, Zuordnung Z
where P.PNr = Z.PNr and
      Budget > 100.000 and
      (Ort = 'MD' or Ort = 'B')
```

■ Selektionsbedingung in KNF:

$P.PNr = Z.PNr \wedge \text{Budget} > 100.000 \wedge (\text{Ort} = 'MD' \vee \text{Ort} = 'B')$

■ Selektionsbedingung in DNF:

$(P.PNr = Z.PNr \wedge \text{Budget} > 100.000 \wedge \text{Ort} = 'MD') \vee$
 $(P.PNr = Z.PNr \wedge \text{Budget} > 100.000 \wedge \text{Ort} = 'B')$

Abbildung 106: Beispiel: Normalisierung

7.4 Logische Optimierung

- heuristische Methoden
 - etwa algebraische Optimierung
 - für Relationsalgebra + Gruppierung, ...
- exakte Methoden
 - Tableauoptimierung
 - Anzahl Verbunde minimieren
 - für spezielle Relationsalgebra-Anfragen

7.4.1 Algebraische Optimierung

- Termbesetzung von Termen der Relationsalgebra anhand von Algebraäquivalenzen
- Äquivalenzen gerichtet als Ersetzungsregeln

- heuristische Methode: Operationen verschrieben, um kleinere Zwischenergebnisse zu erhalten; Redundanzen erkennen

```
BÜCHER = { Titel, Autor, Verlag, ISBN }
VERLAGE = { Verlagsname, VerlagsAdr }
ENTLEIHER = { EntlName, EntlAdr, EntlKarte }
AUSLEIHE = { EntlKarte, ISBN, Datum }
```

Abbildung 107: Beispiel

7.4.2 Entfernen redundanter Informationen

- bei Anfragen mit Sichten nötig

$$r(\text{LANGEWEG}) = r(\text{BÜCHER}) \bowtie$$

$$\pi_{\text{ISBN}, \text{DATUM}}(\dots, \sigma_{\text{DATUM} < '31.12.1995'}(r(\text{AUSLEIHE})))$$

- Anfrage an Sicht:

$$\pi_{\text{TITEL}}(r(\text{BÜCHER}) \bowtie r(\text{LANGEWEG}))$$

- Sichtexpansion:

$$\pi_{\text{TITEL}}(r(\text{BÜCHER}) \bowtie r(\text{BÜCHER}) \bowtie \pi_{\dots}(\dots))$$

- Regel: Idempotenz

$r = r \bowtie r$, d.h. \bowtie ist idempotent

Abbildung 108: Entfernen redundanter Informationen

7.4.3 Verschieben von Selektionen

$$\sigma_{\text{AUTOR} = 'Heuer'}(r(\text{BÜCHER}) \bowtie \pi_{\text{ISBN}, \text{DATUM}}(\dots))$$

günstiger:

$$(\sigma_{\text{AUTOR} = 'Heuer'}(r(\text{BÜCHER}))) \bowtie \pi_{\text{ISBN}, \text{DATUM}}(\dots)$$

Abbildung 109: Verschieben von Selektionen

Regel:

Selektion und Verbund kommutieren

nur, wenn die Attribute der Selektionsprädikate dies zulassen.

7.4.4 Reihenfolge von Verbunden

- Kenntnis der Statistikinformationen des Katalogs nötig

$$(r(\text{VERLAGE}) \bowtie r(\text{AUSLEIHE})) \bowtie r(\text{BÜCHER})$$

- erster Verbund: kartesisches Produkt, daher:

$$r(\text{VERLAGE}) \bowtie (r(\text{AUSLEIHE}) \bowtie r(\text{BÜCHER}))$$

Abbildung 110: Reihenfolge von Verbunden

Regel:

- ▣ ist assoziativ und kommutativ

keine eindeutige Vorzugsrichtung bei der Anwendung dieser Regel (daher interne Optimierung, Kostenbasierung)

7.4.5 Einfacher Optimierungsalgorithmus

- komplexe Selektionsprädikate auflösen (Regel **SelSel**), ggf. Regeln der Auflösung für \neg und \vee
- mittels der Regel **SelJoin**, **SeLProj**, **SelUnion** und **SelDiff** Selektion möglichst weit in Richtung der Blätter verschieben, ggf. Selektion gemäß Regel **SelSel** vertauschen
- Verschieben der Projektionen in Richtung der Blätter mittels der Regel **ProjProj**, **ProjJoin**, **ProjUnion**

Einzelschritte werden in der genannten Reihenfolge solange ausgeführt, bis keine Ersetzungen mehr möglich sind.

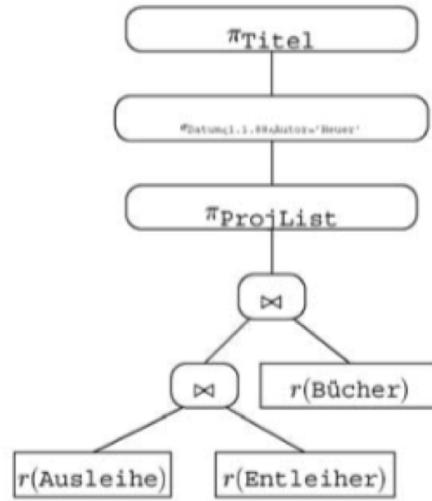


Abbildung 111: Unoptimierter Anfrageplan

Nach Verschiebung der Selektion:

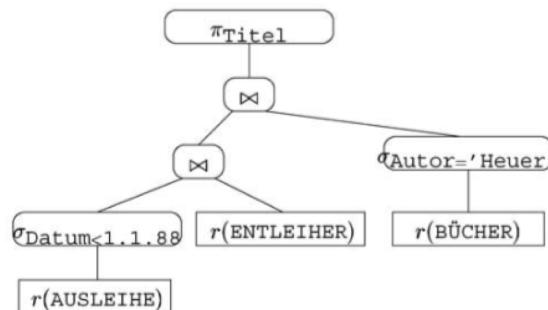


Abbildung 112: Anfrageplan - 1

Mit zusätzlicher Projektion:

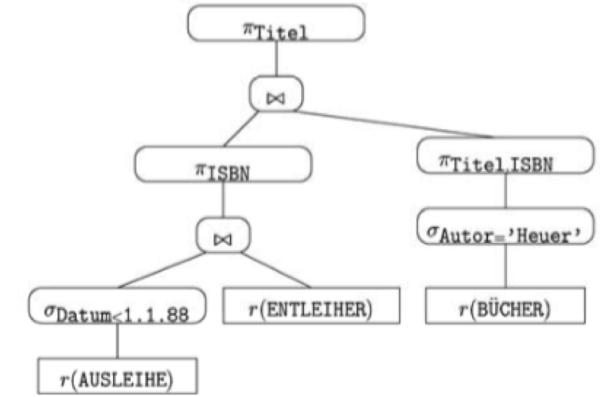


Abbildung 113: Anfrageplan - 2

7.5 Kostenbasierte Auswahl

Berücksichtigung:

- tatsächliche Größe der Datenbankrelationen
- Existenz von Indexen (Primär, Sekundär) und ihre Größe
- Clustering mehrere Relationen
- Selektivität eines Attributs, über das ein Index aufgebaut wurde

Anzahl der verschiedenen Werte des Attributs A in R :

■ Gleichheit:

$$sel(A=c, R) = \frac{1}{V(A, r)}$$

■ Ungleichheit:

$$sel(\text{not } A=c, R) = 1 - sel(A=c, R) = 1 - \frac{1}{V(A, r)}$$

■ Vergleich mittels $<, >, \dots$:

$$sel(A < c, R) = sel(A > c) = \frac{1}{2}$$

Abbildung 114: Selektivität von Attributen - 1

Verfeinerung:

$$sel(\mathbb{A} \leq c, R) = \frac{A_{max} - c}{A_{max} - A_{min}}$$

Bereichsanfragen:

$$sel(c_u \leq \mathbb{A} \leq c_o, R) = \frac{c_o - c_u}{A_{max} - A_{min}}$$

Selektivitäten für Verbunde:

$$sel_{\bowtie}(\varphi, R, S) \approx \frac{|R \bowtie_{\varphi} S|}{|R \times S|}$$

Abbildung 115: Selektivität von Attributen - 2

7.5.1 Selektivitätsschätzung

1. Parametrisierte Funktionen

Parametrisierung einer Funktion, die die Datenverteilung gut widerspiegelt, möglichst genau angeben (etwa Normalverteilung)

2. Histogramme

Wertebereich in Unterbereiche aufteilen und die tatsächlich in diese Unterbereiche fallenden Werte zählen

3. Stichproben

Selektivität anhand einer zufälligen Stichprobe der gespeicherten Datensätze bestimmen

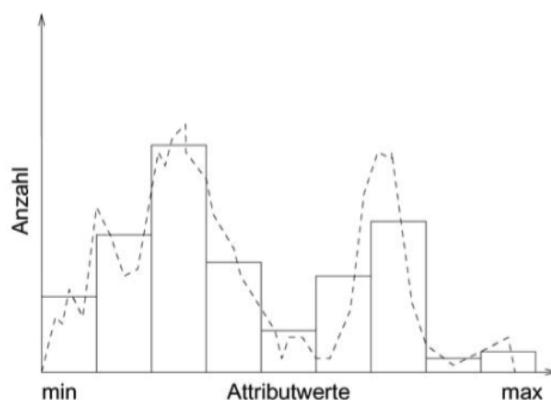


Abbildung 116: Beispiel: Histogramm

7.5.2 Optimierer-Architektur

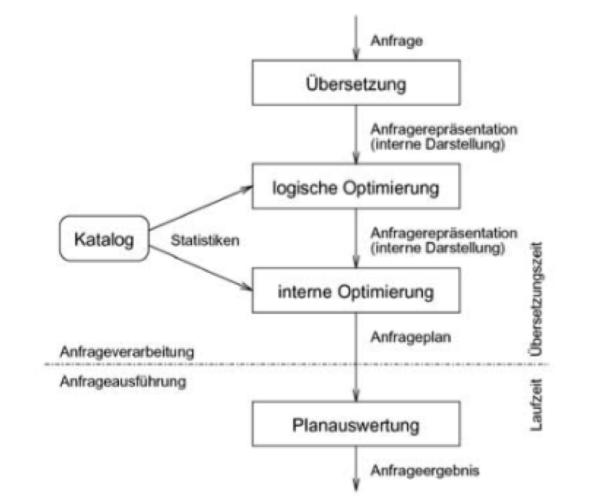


Abbildung 117: Optimierer-Architektur

8 Transaktionen

8.1 Das Transaktionskonzept

8.1.1 Definition

Unter einer *Datenbanktransformation* versteht man einen logische Einheit, die eine oder mehrere *Datenbankzugriffsoperationen* innerhalb eines Anwendungsprogramms umfasst. Dazu gehören *Einfügen*, *Löschen*, *Ändern* und *Abfragen* von Daten.

Die Folge von Operationen einer Transaktion wird häufig durch **begin transaction** eingeleitet und durch **end transaction** abgeschlossen.

Abbruch(Abort) einer Transaktion bedeutet, dass die Abarbeitung vorzeitig durch externe oder interne Einflüsse beendet wird.

8.1.2 Read/Write-Modell

- Datenbankzugriffsoperationen beziehen sich auch identifizierbare **Datenbankobjekte (data items)** z.B. auf ein Feld einer Tabelle, einen Datensatz, ein Datenfeld usw.
- Alle *Operationen* einer Transaktion *auf der Datenbank* lassen sich auf die elementaren Operationen **read item** und **write item** zurückführen.
- Alle *anderen* Operationen einer Transaktion sind für das Transaktionsmanagement innerhalb der Datenbank irrelevant.



Abbildung 118: read/write modell

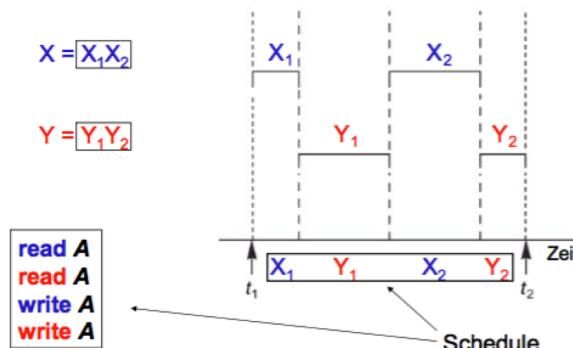


Abbildung 119: Mehrbenutzerbetrieb

8.1.3 Das ACID-Prinzip

- Atomicity (*Ununterbrechbarkeit*)
⇒ Transaktion wird ganz oder gar nicht ausgeführt
- Consistency (Konsistenz oder *Integritätserhaltung*)
⇒ der Zustand der von einer Transaktion hinterlassen wird genügt den Integritätsbedingungen
- Isolation
⇒ Bei mehrfachem Zugriff auf DB soll dieser Zugriff isoliert von allen anderen ablaufen
- Durability (*Dauerhaftigkeit* oder *Persistenz*)
⇒ Nach Ende einer Transaktion stehen Ergebnisse dauerhaft in der DB

8.1.4 Transaktionsverwaltung

Transaktionssteuerung

Commit

- signalisiert, dass alle seit Beginn der Transaktion durchgeführten Manipulationen an Datenbankobjekten definitiv in der Datenbank abgespeichert werden sollen
- macht Datenbankänderungen (für andere) i.a. erst sichtbar (strikt)

Abort

- Bricht eine Transaktion ab und macht alle bisher durchgeführten Änderungen in der Datenbank wieder rückgängig (Rollback)
- kann von der Anwendung oder durch das Datenbanksystem ausgelöst werden

8.2 Synchronisationsprobleme

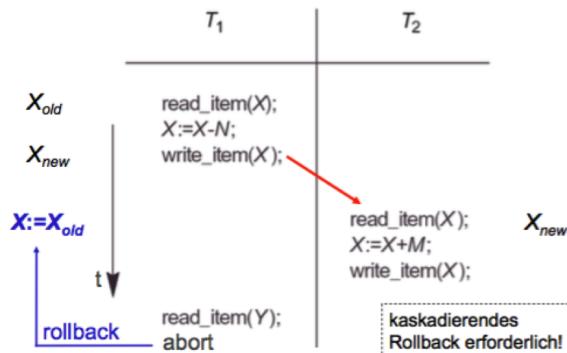


Abbildung 120: Das Dirty-Read-Problem

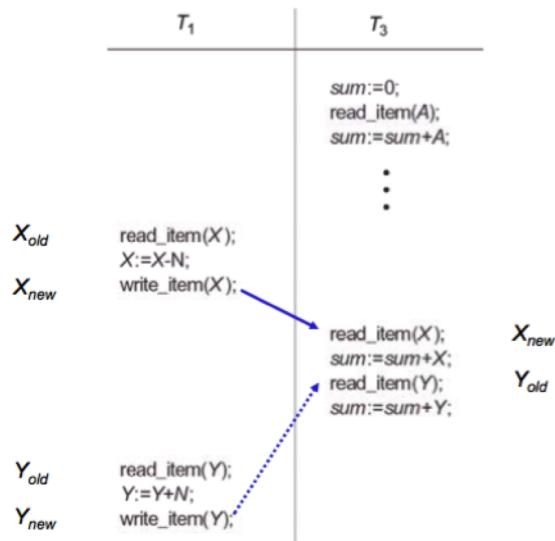


Abbildung 121: Das Incorrect-Summary-Problem

Weitere Probleme

- Unrepeatable-Read-Problem
Eine Transaktion liest zweimal das gleiche Datenobjekt und erhält unterschiedliche Werte
 - Phantom-Problem
Eine Transaktion liest mehrfach eine Menge von Datensätzen und erhält bestimmte Sätze nicht mehr bzw. erhält temporär weitere Sätze.
- Zur Vermeidung der genannten Probleme unterscheidet man verschiedene Stufen der Isolation von Transaktionen

read uncommitted (read only), **read committed**, **repeatable read**, **serializable**

8.3 Schedules und Serialisierbarkeit

8.3.1 Definition: Schedule

Unter einem **Schedule S** für n Transaktionen T_1, \dots, T_n versteht man eine (meist totale) Ordnung aller ihrer Operationen mit der Bedingung, dass die Abfolge der Operation für jedes T_i in S erhalten bleibt, aber durch Operationen einer oder mehrerer anderer Transaktion T_j unterbrochen sein kann.

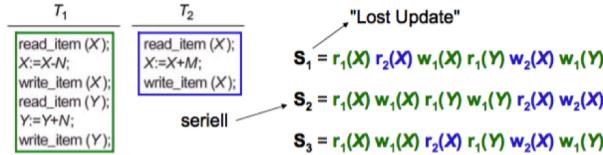


Abbildung 122: Schedules

Konflikte Zwei Operationen stehen im **Konflikt** miteinander, wenn sie

- zu unterschiedlichen Transaktionen gehören und
- auf das gleiche Datenobjekt zugreifen und
- mindestens eine von ihnen eine write-Operation ist.

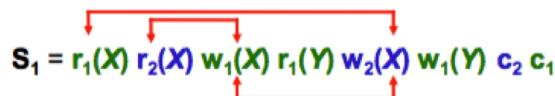


Abbildung 123: Schedules - Konflikte

Ein Schedule ist vollständig, wenn er für jede Transaktion jeweils am Ende eine Commit- oder Abort-Operation enthält, und für je zwei in Konflikt stehende Operationen klar ist, welche zuerst ausgeführt werden soll.

8.3.2 Serialisierbarkeitsbegriff

Das **Ergebnis** eines Schedules ist der aus der Auswertung der Operation in der gegebenen Reihenfolge resultierende Datenbankzustand.

Grundsatz: Ergebnis eines seriellen Schedules ist **korrekt**

Definiton Ein Schedule S für n Transaktionen T_1, \dots, T_n ist **serialisierbar**, wenn er äquivalent zu einem seriellen Schedule für T_1, \dots, T_n ist.

Wie definiert man Äquivalenz?

8.3.2.1 Ergebnisäquivalenz

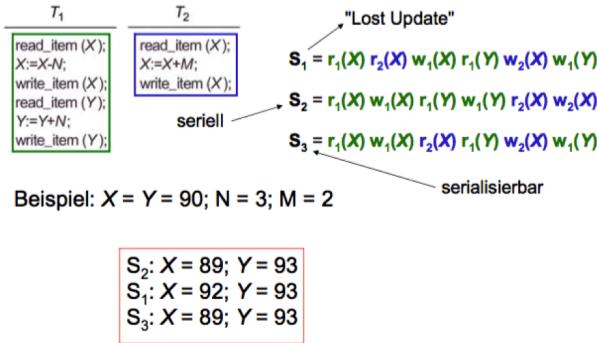


Abbildung 124: Ergebnisäquivalenz

8.3.3 Konfliktäquivalenz und -serialisierbarkeit

Zwei Schedules sind konfliktäquivalent, wenn für alle Paar von Operationen, die in Konflikt stehen, die Reihenfolge in beiden Schedules gleich ist.

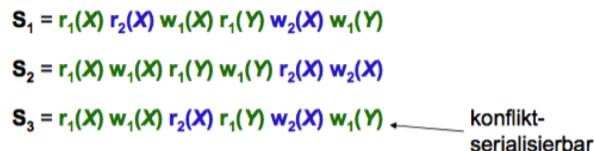


Abbildung 125: Konfliktserialisierbar

Definiton Ein Schedule S für n Transaktionen T_1, \dots, T_n ist **konfliktserialisierbar**, wenn er **konfliktäquivalent** zu einem seriellen Schedule für T_1, \dots, T_n ist.

8.4 Serialisierbarkeitstest

Serialisierbarkeit eines Schedules lässt sich einfach testen durch Aufbau eines **Vorgänger-/Nachfolgergraphen** der beteiligten Transaktionen:

- Jede Transaktion T entspricht einem Knoten T
- Stehen in einem Schedule S bei zwei Transaktionen T_1 und T_2 Operationen $O_1(X)$ und $O_2(X)$ auf dem gleichen Datenobjekt X in Konflikt und tritt $O_1(X)$ in S vor $O_2(X)$ auf, dann enthält der Graph eine gerichtete Kante von T_1 nach T_2 .

Ein Schedule S ist (konflikt-)serialisierbar, falls der zugehörige **Vorgänger-/Nachfolgergraph** **keinen Zyklus** aufweist.

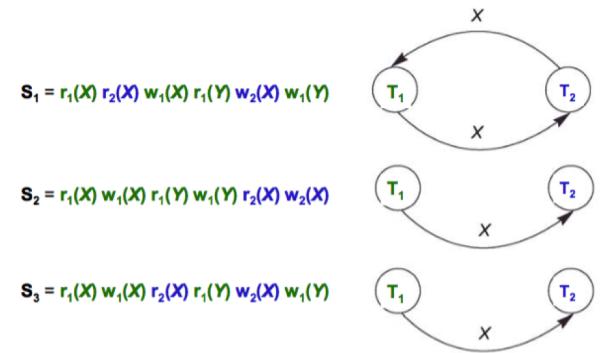


Abbildung 126: Beispiel: Vorgänger-/Nachfolgergraphen

8.5 Synchronisation mit Sperrverfahren

8.5.1 Synchronisationsprotokolle

Um die Serialisierbarkeit von Schedules zu garantieren, benötigt ein Datenbanksystem ein Protokoll, mit dem es die Transaktionen analysiert und eine geeignete Abfolge aller Operationen ermittelt.

- Sperrverfahren
- Zeitstempelverfahren
- Mehrversionsverfahren
- Optimisitsche Verfahren

8.5.2 Typen von Sperren

Grundidee Mit jedem Datenobjekt verbindet man eine Variable, in der vermerkt ist, ob es für eine bestimmte Operation zugereifbar ist. Diese Variable wird in einer **Sperroperation**, die der eigentlichen Operation vorangeht, gelesen und geschrieben.

- binäre Sperren:
 - `lock(X)` und `unlock(X)`
- Schreib-/Lesesperren:
 - `read_lock(X)`, `write_lock(X)`, `unlock(X)`

Der *Lock-Manager* eines Datenbanksystems verwaltet den Typ und die Sperren pro Datenobjekt in einer Sperrtabelle.

```
unlock_item (X):
    if LOCK (X)="write-locked"
        then begin LOCK (X)← "unlocked;"
            wakeup one of the waiting transactions, if any
        end
    else if LOCK(X)="read-locked"
        then begin
            no_of_reads(X)← no_of_reads(X) - 1;
            if no_of_reads(X)=0
                then begin LOCK (X)="unlocked";
                    wakeup one of the waiting transactions, if any
                end
            end;
        end;
```

Abbildung 127: Sperroperationen

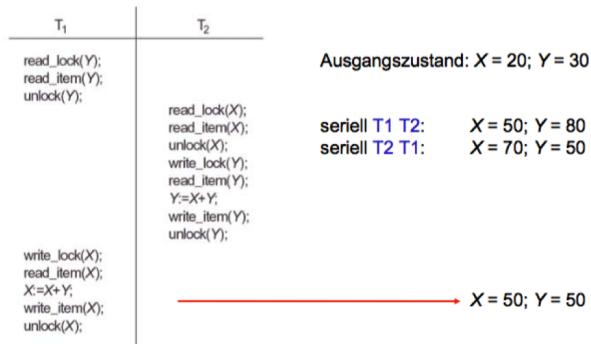


Abbildung 128: Sperren und Serialisierbarkeit

8.6 Zweiphasensperren (2PL)

Das *Sperrprotokoll* muss zur Sicherstellung der *Serialisierbarkeit* die für eine Transaktion benötigten Sperroperationen in **zwei Phasen** ausführen:

- in der **Wachstumsphase** werden alle Sperren gesetzt, die für Lese- und Schreiboperationen erforderlich sind
- in der **Schrumpfungsphase** werden alle Sperren wieder freigegeben

Die erste Freigabe einer Sperre erfolgt also nach dem letzten Setzen einer Sperre *unabhängig vom Sperrobjekt*

Schedules mit Transaktionen, die 2PL unterliegen sind serialisierbar

8.6.1 2PL-Varianten

2PL wie definiert bezeichnet man als **Basis-2PL**

- Beim **konservativen** 2PL werden alle Sperren zu Beginn der Transaktion, also vor der eigentlichen Ausführung angefordert. Die Sperren werden nur gesetzt, wenn sie *alle* verfügbar sind
- **Stricktes** 2PL gibt *Schreibsperren* erst nach dem Commit oder Abort frei. Die entsprechenden Schedules sind strikt, d.h. es können von anderen Transaktionen nur Ergebnissen von (erfolgreich) beendeten Transaktionen gelesen (oder überschrieben) werden (vgl. Dirty-Read)
- Beim **rigorosem** 2PL gilt diese Bedingung auch bereits für Lesesperrungen

8.6.2 Verklemmungen (Deadlocks)

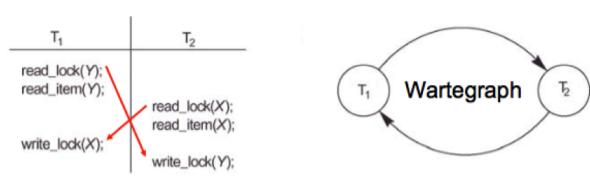


Abbildung 129: Deadlocks

2PL ist im allgemeinen nicht Deadlock-frei!

Ausnahme: konservatives 2PL

Deadlock **Vermeidung** in Verbindung mit 2PL ist zu aufwendig, man beschränkt sich daher auf die Deadlock **Entdeckung** mit Hilfe der Wartgraphen.

9 Recovery und Datensicherung

9.1 Fehlerklassifikationen

1. Transaktionsfehler
2. Systemfehler
3. Mediafehler

→ unterschiedliche Recovery-Maßnahmen je nach Fehlerart.

9.1.1 Transaktionsfehler

- Haben den Abbruch der jeweiligen Fehlerklasse zur Folge
- haben keinen Einfluss auf den Rest des Systems: → auch lokaler Fehler

Typische Transaktionsfehler:

1. Fehler im Anwendungsprogramm
2. Transaktionsabbruch explizit durch den Benutzer
3. Transaktionsabbruch durch das System

Behandlung:

→ “Isoliertes” Zurücksetzen aller Änderungen der abgebrochenen Transaktionen

9.1.2 Systemfehler

- Folge: Zerstörung der Daten im Hauptspeicher
- betreffen jedoch nicht den Hintergrundspeicher

Typische Systemfehler:

1. DBMS-Fehler
2. Betriebssystemfehler
3. Hardwarefehler

Behandlung:

→ Zurücksetzen der von **nicht beendeten** Transaktionen in die DB eingebrachten Änderungen

→ Nachvollziehen der von abgeschlossenen Transaktionen **nicht in die DB eingebrachten** Änderungen

9.1.2.1 Szenario: Systemfehler

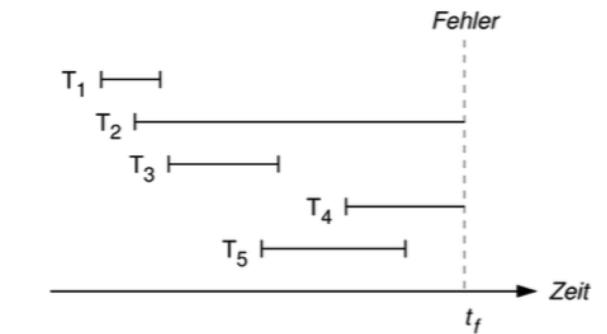


Abbildung 130: Systemfehler

Folgen:

- Inhalt des flüchtigen Speichers zum Zeitpunkt t_f ist unbrauchbar → Transaktionen in unterschiedlicher Weise davon betroffen

Transaktionszustände:

- zum Fehlerzeitpunkt noch aktive Transaktionen (T_1 und T_4)
- bereits vor dem Fehlerzeitpunkt beendete Transaktionen (T_1, T_3 und T_5)

Probleme:

- Dauerhaftigkeitseigenschaft ⇒ Effekte von T_1, T_3 und T_5 **müssen** dauerhaft in der DB sein
- Atomaritätseigenschaft ⇒ Effekte von T_1 und T_4 **dürfen** nicht in der DB sein

9.1.3 Mediafehler

- Ziehen den Verlust von stabilen Datenbankdaten nach sich

Ursachen:

1. "Head-Caches"
2. Controller-Fehler
3. Naturgewalten wie Feuer oder Erdbeben

Maßnahmen:

→ DB-Archiv auf anderen "Medien"

9.2 Recovery-Klassen

- **R1-Recovery (lokales Zurücksetzen - Transaction UNDO)**
nach Transaktionsfehler werden die entsprechenden Transaktionen isoliert zurückgesetzt
- **R2-Recovery (partielles Wiederholen - Partial REDO)**
nach Systemfehler besteht ein konsistenter Zielzustand aus allen bis zum Fehler abgeschlossenen Transaktionen
⇒ alle Änderungen abgeschlossener Transaktionen, deren Daten beim Systemfehler noch im Puffer waren nachvollziehen und in die DB schreiben
- **R3-Recovery (globales Zurücksetzen - Global UNDO)**
nach Systemfehler soll der Zielzustand keine Auswirkungen nicht beendeter Transaktionen enthalten
⇒ Spuren sämtlicher zum Fehlerzeitpunkt aktiver Transaktionen aus der DB entfernen.
- **R4-Recovery (globales Wiederholen - Global REDO)**
nach Defekt auf einem nichtflüchtigen Externspeicher wird eine Archivkopie auf den Datenträger kopiert
⇒ alle Änderungen der nach der letzten Erstellung der Archivkopie beendeten Transaktionen nachvollziehen und in die DB schreiben

9.3 Protokollierungsarten

9.3.1 Log-Buch

Schritt	T ₁	T ₂	Log
1	lock A		(T ₁ , begin)
2	read A		
3	A := A - 1		
4	write A		(T ₁ , A, 10, 9)
5	lock B		
6	unlock A		
7		lock A	(T ₂ , begin)
8		read A	
9		A := A × 2	
10	read B		
11		write A	(T ₂ , A, 9, 18)
12		commit	(T ₂ , commit)
13		unlock A	
14	B := B/A ↓		(T ₁ , abort)

Abbildung 131: Aufbau des Log-Buchs

```
[ LSN, TA, PageID, Redo, Undo, PrevLSN ]
```

- **LSN:** Log-Sequence-Number, eindeutige und aufsteigende Durchnumerierung der Log-Einträge
- **TA:** Transaktionskennung
- **PageID:** Seitennummer
- **Redo:** REDO-Information
- **Undo:** UNDO-Information
- **PrevLSN:** Verweis auf den vorherigen Eintrag der selben Transaktion

Abbildung 132: Log-Einträge

```
[ #1, T1, BOT ]
[ #2, T1, PA, A=A+1, A=A-1, #1 ]
[ #3, T2, BOT ]
[ #4, T2, PA, A=A×2, A=A/2, #3 ]
[ #5, T2, commit, #4 ]
[ #6, T1, abort, #2 ]
```

Abbildung 133: Beispiel: Log-Einträge

9.3.2 Physisches Protokollieren

- ganze physische Speichereinheiten (d.h. Seiten)
- vor dem Einlagern Seiten gesondert als **Before-Image** speichern

Vorteil:

- Protokoll- bzw. Recovery-Komponenten sind sehr einfach zu realisieren

Nachteile:

- Protokollinformationen können nicht gepuffert werden → hoher E/A-Aufwand
- Seitenprotokollierung erfordert das Sperren ganzer Seiten
- Protokollinformationen über Änderungen in den Zugriffspfaden, Tabellen, Ketten, etc. müssen zusätzlich gehalten werden
- mengenwertige Änderungen (bulk updates) erfordern umfangreiche Protokollierung

9.3.3 Logisches Protokollieren

- alle ausgeführten höheren Operationen werden im Log-Buch erfasst
→ anhand dieser Informationen können die DML-Anweisungen (und deren Invers-Operationen) nachvollzogen werden

Vorteil:

- Auswirkungen der Änderungsoperationen einer Transaktion auf die Speicherungsstrukturen müssen nicht protokolliert werden
→ es genügt Änderungsoperationen und die aktuellen Parameter zu notieren

Nachteile:

- Probleme bei der R1/R3-Recovery → inverse DML-Operationen sind oftmals nicht (trivial) berechenbar
- DM muss in einem specherkonsistenten Zustand sein, um als Ausgangspunkt für die Recovery dienen zu können

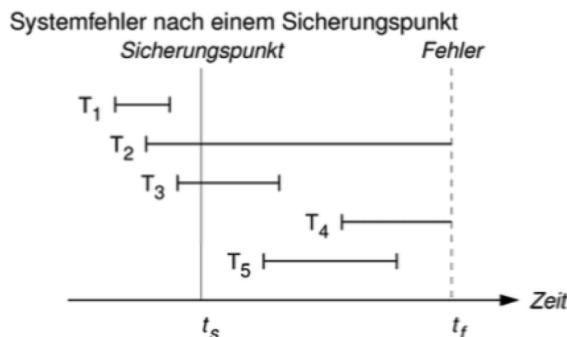


Abbildung 134: Szenario

	Phys. Protokollieren	Log. Protokollieren
T ₁	In t_s wurden alle bis dahin angefallenen Änderungen übernommen (keine Wiederholung von T ₁ notwendig)	
T ₂	Partielles Zurücksetzen von T ₂ mit Hilfe der Before-Images	Ausführung bis t_s protokollierter inverser DML-Befehle in umgekehrter Reihenfolge bis BOT
T ₃	Partielles Wiederholen von T ₃ mit Hilfe der After-Images	Ausführung nach t_s protokollierter Original-DML-Befehle bis Commit
T ₄	Alle Effekte von T ₄ mit Wiederherstellung des Zustands zum Zeitpunkt t_s implizit entfernt (keine weiteren Maßnahmen erforderlich)	
T ₅	Durch Wiederherstellen des Zustands zum Zeitpunkt t_s verschwinden alle Auswirkungen von T ₅ (T ₅ komplett wiederholen)	

Abbildung 135: physisches vs. logisches Protokollieren

9.3.4 Das WAL - Prinzip

WRTIE AHEAD LOG

- vor **commit** einer Transaktion Ausschreiben aller zugehörigen Log-Einträge (notwendig für Durchführung von *REDO*)
- vor dem Auslagern einer modifizierten Seite Schreiben aller zugehörigen Log-Einträge in das Log-Archiv (ermöglicht *UNDO* bei abgebrochenen Transaktionen)

kontinuierliches Schreiben auf Ringpuffer.

9.4 Recovery-Strategien

9.4.1 Seitenersetzungsstrategien

UNDO (steal)

- jederzeit dürfen noch nicht freigegebene Seiten ausgelagert werden
 - benötigt das **Write-Ahead-Logging-Protokoll**
 - Sicherung von Protokollinformationen vor Seitenauslagerung

NO-UNDO (\neg steal)

- kein Auslagern von geänderten Seiten vor dem Commit einer Transaktion erlaubt
 - vermeiden das Zurücksetzen von Transaktionen
 - vereinfacht den Abbruch einer Transaktion
 - hat Probleme, wenn keine der im Puffer modifizierten Seiten ausgelagert werden dürfen

9.4.2 Propagierungsstrategien

NO-REDO (force)

- beim Commit werden alle geänderten Seiten in die DB eingebracht

REDO (\neg force)

- nach dem Commit können geänderte Seiten im Puffer verbleiben, ohne explizit auf dem stabilen Speicher gespeichert werden
 - Redo-Protokollinformationen im stabilen Speicher abgelegt

Vergleich:

- REDO-Variante ist im allgemeinen besser, weil
 - sie den großen E/A-Aufwand beim Commit und damit schlechte Antwortzeiten vermeidet
 - sie durch den Einsatz von **Sicherungspunkten** verbessert werden kann

9.4.3 Einbringstrategien

Direkte Zuordnung (\neg atomar = update-in-place) :

- jede Seite im Puffer ist genau einer Seite in der DB zugeordnet
 - Puffer-Seite wird beim Auslagern auf die entsprechende DB-Seite kopiert
 - der alte Zustand geht verloren
 - ⇒ erfordert physisches Protokollieren

Indirekte Zuordnung (atomar) :

- jede Puffer-Seite ist im stabilen Speicher ein **Twin-Block** reserviert
 - Puffer-Seite wird jeweils auf den “älteren” Twin-Block ausgelagert
 - selbst bei einem Fehler bleibt der letzten konsistente Zustand erhalten

Nachteil (der indirekten Zuordnung):

1. doppelter Speicherplatzbedarf
2. Seitentabellen für die Abbildung zwischen flüchtigen und stablien Speicher passen evtl. nicht in den Hauptspeicher

9.4.4 Konkrete Recovery-Strategien

Kombination der Seitenersetzungs - und Propagierungsstrategien ergeben die möglichen Recoverystrategien:

1. UNDO / REDO
2. UNDO / NO-REDO
3. NO-UNDO / REDO
4. NO-UNDO / NO-REDO

Seitenersetzung	Propagierung	
	force	\neg force
\neg steal	kein REDO kein UNDO	REDO kein UNDO
steal	kein REDO UNDO	REDO UNDO

Abbildung 136: Überblick