

# Zusammenfassung IDB

Marco Ammon

13. Februar 2019

## 1 Einführung

- *Datenabstraktion* / *Datenunabhängigkeit*: persistentes Speichern und Wiedergewinnen (Auffinden und Aushändigen) von Daten *unabhängig* von Details der Speicherung
- *Schicht*: realisiert *Dienst* und stellt ihn per *Schnittstelle* zur Verfügung

## 2 Dateiverwaltung

- *physische* Speichergeräte (z.B. Festplatten) werden durch *logische* abstrahiert (z.B. Neueinlesen bei Checksum-Fehlern)
- *Block* als kleinste Einheit der IO
- „Adresse“ eines Blocks: (Zylinder, Spur, Sektor)
- *Dateien* als benannte Menge von Blöcken
- *blockorientierte* Zugriffsmethode: verwendet eindeutige, fortlaufende Blockadressen innerhalb der Datei

## 3 Sätze

- *Satz* als zusammengehörende Daten eines Gegenstands der Anwendung (z.B. Tupel, Objekt) mit variabler oder fester Länge
- *Satzdatei* als Sammlung von Sätzen, kann über verschiedene Blöcke verteilt sein
- Ausprägungen:
  - *sequentiell*:
    - \* Reihenfolge der Abspeicherung und des Auslesens bereits mit Schreiben festgelegt
    - \* *keine* Änderungen / Löschen möglich
    - \* kein wahlfreier Zugriff
  - *direkt*:
    - \* Verwendung sogenannter *Satzadressen* (hier als *TIDs* realisiert; *eindeutig* und *unveränderlich*) als Adresstupel (Block, Index)
    - \* Abbildung von Index auf Offset innerhalb eines Blockes durch Array am Ende eines Blockes
    - \* erlaubt wahlfreien Zugriff
    - \* erlaubt Löschen von Sätzen: Index wird ungültig markiert, folgende Sätze nach vorne verschoben, Anpassung der Offsets
    - \* erlaubt Ändern von Sätzen:

- *ohne* Überlauf: Verschieben der folgenden Sätze, Anpassung der Offsets
- *mit* Überlauf: Satz wird in anderen Block verschoben, Verweis auf diesen wird angelegt, (evtl.) Anpassung der Offsets

## 4 Schlüssel

- *Schlüsselwerte* als „inhaltsbezogene Adressen“
- *Hashing*:
  - *Hash-Funktion* verteilt Schlüsselwert möglichst gleichmäßig auf verfügbare *Buckets* (Blöcke)
  - *Divisions-Rest-Verfahren*:  $h(k) = (k \bmod q)$  mit Schlüsselwert  $k$  und Anzahl der Buckets  $q$
  - Problem des *Überlaufs* mit verschiedenen Lösungsmöglichkeiten:
    - \* *Open Addressing*: Ausweichen auf Nachbarbuckets
    - \* spezielle *Overflow-Buckets*: Bucket verweist auf „*seinen*“ Overflow-Bucket
  - *virtuelles Hashing* zur konstanten Reorganisation:
    - \* Anzahl der Buckets  $q$ , Sätze pro Bucket  $b \Rightarrow$  Kapazität  $:= q \cdot b$
    - \* Belegungsfaktor  $\beta := \frac{\text{Anzahl gespeicherter Sätze } N}{\text{Kapazität}}$
    - \* Wenn  $\beta >$  Schwellwert  $\alpha$ , Menge der Buckets vergrößern
    - \* als *VH1*:
      - Anzahl der Blöcke direkt verdoppeln
      - neue Hashfunktion  $h_2$  einführen
      - Bitmaske um Verwendung der neuen Hashfunktion zu verwalten
      - bei *Einfügen* eines Satzes in ein „altes“ Bucket Neuverteilung dieses Buckets mittels  $h_2$ , Bit setzen
    - \* als *Lineares Hashing*:
      - Positionszeiger  $p$
      - ein neues Bucket anlegen
      - Bucket an Stelle  $p$  mit  $h_2$  aufteilen,  $p++$
      - wenn  $h_1(k) < p$ , dann mittels  $h_2$  verteilen
- *Indizes* mittels *Bäumen*:
  - *B-Baum*:
    - \* jeder *Knoten* ist genau einen Block groß
    - \* *balanciert*, alle Blätter außer Wurzel immer mindestens zur Hälfte gefüllt
    - \* *Knoten*:
      - Anzahl der verwendeten Einträge  $n$ , es gilt  $k \leq n \leq 2k$
      - *Eintrag*: Tupel (Schlüsselwert, Datensatz, Blocknummer des Kindknotens)
      - Einträge nach Schlüsselwert *sortiert*
    - \* Einfügen: wie Suchen; nur in Blattknoten; bei Überlauf „linke“ und „rechte“ Einträge als neue Knoten, „mittlerer“ als *Diskriminator* in Eltern-Knoten einfügen
    - \* Löschen von Schlüssel  $S$  im Blattknoten:
      - Entfernen und ggf. Unterlauf behandeln
    - \* Löschen von Schlüssel  $S$  in innerem Knoten:
      - betrachte die Blattknoten mit direktem Vorgänger  $S'$  und direktem Nachfolger  $S''$  von  $S$

- wähle den größeren
- ersetze  $S$  je nach Wahl durch  $S'$  bzw.  $S''$
- lösche entsprechenden Schlüssel  $S'$  bzw.  $S''$  und ggf. Unterlauf behandeln
- \* Höhe:
  - obere Schranke:  $h(n) = \log_{k+1} \left( \frac{n+1}{2} \right) + 1$
  - untere Schranke:  $h(n) = \log_{k+1} (k + 1)$
- *B\*-Baum* / *B+-Baum*:
  - \* Sätze stehen *ausschließlich* in Blattknoten
  - \* innerer Knoten:
    - Anzahl der verwendeten Einträge  $n$
    - Eintrag: Tupel (Referenzschlüssel, Blocknummer des Kindknotens)
  - \* Blattknoten:
    - Anzahl der verwendeten Einträge  $n$
    - Vorgänger-Zeiger, Nachfolger-Zeiger
    - Eintrag: Tupel (Schlüsselwert, Datensatz)
  - \* Löschen ohne Unterlauf: lösche Satz aus Blatt; Diskriminator muss *nicht* geändert werden
  - \* Löschen mit Unterlauf:
    - Ist Anzahl der Einträge des Blatts und eines Nachbarknotens größer als  $2k$ , verteile Sätze neu auf beide Knoten
    - ansonsten mische beide Blätter zu einem einzigen
- *R-Baum*:
  - \* ähnlich zu B-Baum
  - \* multidimensional
  - \* arbeitet mit Rechtecken
  - \* beim Einfügen Rechteck nur möglichst gering vergrößern
- Müssen nicht zwangsläufig zur *Primärorganisation* verwendet werden, können als „Sätze“ z.B. auch nur Satzadressen enthalten
- *Bitmap-Indizes*: eine Bitmap *pro Schlüsselwert*

## 5 Puffer

- Hauptspeicherbereich, der Blöcke aufnehmen kann, um (Lese-/Schreibe-) Zugriffe zu *beschleunigen*
- *Ersetzungsstrategie*: „Welcher Block wird verdrängt?“
  - *first in, first out* (FIFO): „ältester“ Block
  - *least frequently used* (LFU): am seltensten benutzter Block
  - *least recently used* (LRU): am längsten nicht mehr benutzter Block
    - \* Stacktiefenverteilung: „Wie tief liegen die referenzierten Seiten?“
  - *second chance* (CLOCK): Approximation von LRU mit einfacherer Implementierung:
    - \* Jeder Block im Puffer besitzt ein *Benutzt-Bit*
    - \* bei Verdrängung Suche mit Zeiger
    - \* falls Benutzt-Bit 1, auf 0 setzen und Zeiger weiterschieben

- \* falls Benutzt-Bit 0, Block ersetzen
- *Working Set Size*  $|W(t, w)|$ : Anzahl der unterschiedlichen referenzierten Seiten in den letzten  $w$  Zugriffen bis Zeitpunkt  $t$
- *aktuelle Lokalität*:  $AL(t, w) = \frac{|W(t, w)|}{w}$
- *durchschnittliche Lokalität*:  $L(w) = \frac{\sum_{t=w}^n AL(t, w)}{n-w+1}$
- Zustand im Fehlerfall hängt unter anderem von *Einbringstrategie* (siehe Recovery) und *Seitenzuordnung* ab
- Seitenzuordnung: „Welche Blöcke (in einer Datei) gehören zu einer Seite (im Puffer)?“
  - *direkt*: aufeinander folgende Seiten werden auf aufeinander folgende Blöcke einer Datei abgebildet
  - *indirekt*: *Page Table* enthält zu jeder Seite eine Blocknummer
- Seiteneinbringung:
  - *direkt*: Bei Verdrängung aus Puffer wird genau der Block überschrieben, aus dem ursprünglich eingelagert wurde („update-in-place“)
  - *indirekt*: Bei Verdrängung aus Puffer wird in einen freien Block geschrieben.
- Problem der indirekten Seiteneinbringung: „Wann können alte Blöcke gelöscht werden?“; verschiedene Lösungsansätze:
  - *Schattenspeicher*:
    - \* Änderungen nur auf Kopien, die periodisch dann mit „gesicherter“ Version vertauscht wird
  - *Twin Slots*:
    - \* jede Seite hat zwei Blöcke
    - \* immer beide lesen, bei Änderungen älteren überschreiben

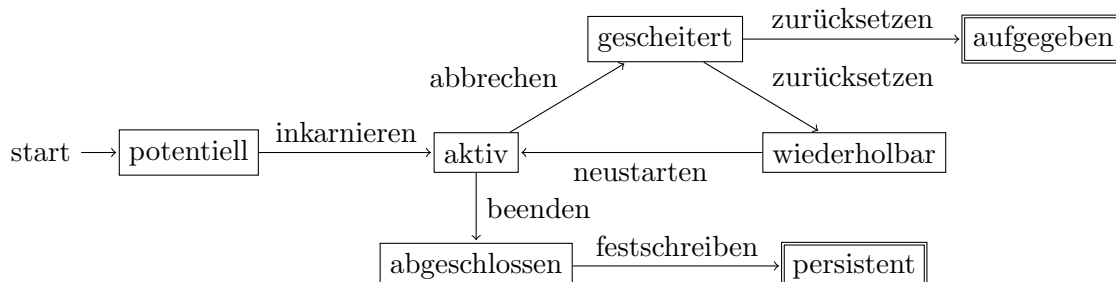
## 6 Programmzugriff

- *Precompiler* übersetzt SQL-Anweisungen (mittels EXEC SQL gekennzeichnet) zur *Compiler-Zeit* in die verwendete Programmiersprache
  - Deklaration der verwendeten Variablen am Anfang mittels DECLARE SECTION
  - Fehlermeldungen und ähnliches werden über die sogenannte *SQL communication area* verwaltet (INCLUDE SQLCA am Anfang)
  - Mengen-orientes Paradigma des DBVS oft nicht mit Programmiersprache vereinbar  $\Rightarrow$  Einrichtung eines *Cursors* zum tupelweisen Durchlaufen der Ergebnismenge
  - Beispielablauf: DECLARE CURSOR  $\rightarrow$  OPEN  $\rightarrow$  (mehrfach) FETCH bis Fehlercode == 100  $\rightarrow$  CLOSE
  - kann durch Präprozessort direkt als *stored procedure* angelegt werden
- *Unterprogrammaufruf (Call-Level-Interface)*:
  - Übergabe der SQL-Anweisungen zur *Laufzeit*
  - Beispiel JDBC
    - \* `Connection con = DriverManager.getConnection(URL, USER, PASSWORD);`
    - \* `Statement anweisung = con.createStatement();`
    - `ResultSet ergebnis = anweisung.executeQuery(ANFRAGE);`
    - außerdem: `int executeUpdate(String sql), boolean execute(String sql)`
    - \* `while (ergebnis.next()) int pnr = ergebnis.getInt(1);`

- Bei mehrfacher Ausführung der gleichen Abfrage mit unterschiedlichen Werten *prepared statements* sinnvoll:
  - \* Anfrage enthält Platzhalter für Werte
  - \* Analyse, Ausführungsplanerstellung und weiteres wird sofort durchgeführte
  - \* JDBC:
    - `PreparedStatement prep = con.prepareStatement(INSERT ...VALUES (?,?))`
    - Setzen der Werte mittels `void setDATENTYP(int paramId, DATENTYP val)`
    - Ausführung mit `prep.executeUpdate()`
- bei stored-procedures nur noch einmaliges Analysieren, etc. zur Compile-Zeit erforderlich:
  - \* Prozedur in DBVS bekommt „Namen“, über den sie mit Werten als Parametern aufrufbar ist
  - \* JDBC:
    - `CallableStatement call = con.prepareCall("{ call PROZEDUR }");`
    - Eingabe-Parameter analog zu prepared statements
    - Ausgabe-Parameter mittels `registerOutParameter(int paramId, int type)`
- *O/R-Mapping* bildet Objekte der Programmiersprache (meist durch Annotationen) auf Tupel der relationalen DB ab

## 7 Transaktionen

- sinnvoll für *nebenläufigen* Zugriff
- erleichtern Umgang mit *Fehlern* und Ausfällen (siehe Recovery)
- *Transaktion* als *logische Einheit* einer Folge von DB-Operationen (von einem logisch konsistenten Zustand zum nächsten):
  - bei Fehler vor Ende: Rückgängigmachen der bisher durchgeführten Änderungen
  - bei Fehler nach Ende: kein Problem
  - Anfang meist implizit (oder *begin*)
  - Ende durch `commit` (Änderungen sollen durchgeführt werden) bzw. *abort* (Änderungen sollen verworfen werden)
- *ACID*-Eigenschaften einer Transaktion:
  - *Atomarität* („alles oder nichts“ wird ausgeführt)
  - *Konsistenz*
  - *Isolation* (gegenüber anderen Zugriffen auf DB)
  - *Dauerhaftigkeit* (auch nach Fehler bleiben erfolgreiche Transaktionen bestehen)
- „Lebenszyklus“ einer Transaktion



- *Anomalien im Mehrbenutzerbetrieb:*
  - *dirty read:* Lesen von nicht freigegeben Änderungen:  $w_1[x], r_2[x]$ , erst danach Commit / Rollback
  - *dirty write:* Überschreiben von nicht freigegebenen Änderungen:  $w_1[x], w_2[x]$ , erst danach Commit / Rollback
  - *non-repeatable read:* Änderung nachdem gelesen wurde:  $r_1[x], w_2[x]$ , erst danach Commit / Rollback
  - *Phantom-Problem:* Ändern/Anlegen eines Tupels, das gelesenes Prädikat  $P$  erfüllt:  $r_1[P], w_1[y \in P]$ , erst danach Commit/Rollback
- *Serialisierbarkeitstheorie:*
  - Ablauf ist *serialisierbar*, wenn es einen *äquivalenten seriellen* Ablauf seiner Transaktionen gibt
  - Äquivalenz von Abläufen  $G, H$ , wenn für jedes Datenobjekt  $A$  gilt ( $< \hat{=}$  „vor“):
 
$$r_i[A] <_H w_j[A] \Leftrightarrow r_i[A] <_G w_j[A]$$

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

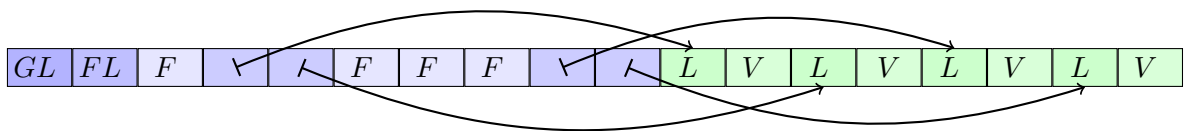
$$w_i[A] <_H w_j[A] \Leftrightarrow w_i[A] <_G w_j[A]$$
  - *Abhängigkeitsgraph* hat *keine* Zyklen  $\Rightarrow$  Ablauf serialisierbar
- *Sperrverfahren mittels Sperrtabelle:*
  - Sperrung muss *vor* Zugriff erfolgen
  - Transaktionen fordern Sperre nicht erneut an
  - Sperren müssen beachtet werden
  - erst am Ende einer Transaktion dürfen Sperren freigegeben werden
  - *X-Sperre:* exklusiv, für Änderungen notwendig
  - *S-Sperre:* geteilt, für Lesen notwendig
  - *IX-Sperre:* exklusiv, zeigt Sperren auf feingranularerer Ebene an
  - *IS-Sperre:* geteilt, zeigt Sperren auf feingranularerer Ebene an
  - *SIX-Sperre:* S + IX, wenn alle Tupel gelesen, aber nur einige geändert werden
  - *Top-Down-Erwerb, Bottom-Up-Freigabe* der Sperren

## 8 Speicherung

- Speicherung der Tupel in Sätzen:
  - zusammengesetzt aus Feldern mit Namen, Typ und Länge (maximal oder variabel)
  - *Metadaten* in *Systemkatalog* gespeichert
  - *Satztyp:* Menge von Sätzen gleicher Struktur (z.B. Tupel einer Relation)
- verschiedene *Speicherungsstrukturen* in Sätzen:
  - mit *eingebetteten Längefeldern:* Gesamtlänge  $GL$ , Inhalt fester Länge  $F$ , zu jedem Inhalt variabler Länge  $V$  vorher die Länge  $L \Rightarrow$  satzinterne Adresse kann *nicht* direkt aus Katalogdaten berechnet werden

GL	F	L	V	L	V	F	F	F	L	V	L	V
----	---	---	---	---	---	---	---	---	---	---	---	---

- eingebettete Längfelder mit *Zeigern*: Länge des festen Strukturteils  $FL$ , Zeiger auf variable Bereiche  $\Rightarrow$  satzinterne Adresse kann aus Katalogdaten berechnet werden



- *spaltenweises* Abspeichern mittels *C-Store*:
  - vor allem auf das Lesen optimiert
  - Änderungen durch Löschen und Einfügen
  - *Projektion*:
    - \* eine oder mehrere Spalten einer Tabelle (und ggf. anderer, über Fremdschlüssel erreichbare Tabellen) nach einem Attribut sortiert
    - \* Speicherschlüssel (*Storage Keys, SK*) für jedes Tupel, aus Position berechenbar
    - \* *Verbund-Indizes (Join Indices)*: Seien T1 und T2 Projektionen der Tabelle T, dann ist ein Join-Index von T1 zu T2 eine Liste von Tupeln aus T2 um den jeweiligen SK aus T1 ergänzt.
  - verschiedene *Komprimierungen* abhängig von Sortierung und Anzahl der verschiedenen Werte:
    - \* sortiert, wenige verschiedene Werte: Tripel (Wert, Position des ersten Auftretens, Anzahl des gleichen Werts) in B-Baum
    - \* unsortiert, wenige verschiedene Werte: laulängenkodierte Bitmaps pro Wert mit B-Baum zum Auffinden der richtigen Bitmap
    - \* sortiert, viele verschiedene Werte: Delta-Kodierung (Differtenz zum Vorgänger) mit B-Baum als Primärorganisation
    - \* unsortiert, viele verschiedene Werte: unkomprimiert, bei Zeichenketten *Dictionary*

## 9 Anfrageverarbeitung

- Abbildung von mengenorientierten Operatoren auf interne Satzchnittstelle
- *Anfrageverarbeitung* erstellt einen *Anfrageausführungsplan*:
  - Analyse: lexikalische und syntaktische Prüfung, semantische Prüfung, Zugriffskontrolle, Integritätskontrolle
  - Optimierung:
    - \* *Standardisierung* und Vereinfachung
    - \* *algebraische* Verbesserung
    - \* *nicht-algebraische* Verbesserung: Berücksichtigung der Kosten der *Planoperatoren*
  - Code-Generierung
  - Ausführungskontrolle
- *logische Operatoren* mit Relationen  $R$ ,  $S$  und Prädikat  $P$ :
  - *Selektion*  $SEL(R, P)$
  - *Projektion*  $PROJ(R, L)$  mit  $L = (A_1, \dots, A_k)$
  - *Kreuzprodukt*  $CROSS(R, S)$
  - *Verbund*  $JOIN(R, S, P(R_{A_i}, S_{A_j}))$
  - *Vereinigung*  $UNION(R;S)$
  - *Schnitt*  $INTERSECT(R,S)$

- *Ausschluss* EXCEPT(R,S)
  - analoge Operationen auf *Multimengen*
  - *Umbenennung* RENAME( $R, R_{\text{neu}}, ((A_i, A_{i,\text{neu}}), \dots)$ )
  - *Duplikat-Eliminierung* DUP-ELIM( $R$ )
  - *Aggregation* SUM( $R, A_i$ ), AVG( $R, A_i$ ), MIN( $R, A_i$ ), MAX( $R, A_i$ ), COUNT( $R$ )
  - *Gruppierung* GROUP( $R, L, G$ ) mit  $G = ((\text{AGG}_1, (A_i), \text{name}_1), \dots)$
  - *erweiterte Projektion* G-PROJ( $R, L$ ) mit  $L = (\text{name}_1 = \text{expr}_1, \dots)$
  - *Sortierung* SORT( $R, L$ ) mit  $L = (A_i, A_j, \dots)$
  - *äußerer Verbund* OUTER-JOIN( $R, S, P, c$ ) mit  $c \in \{\text{left, right, full}\}$
- allgemeine Vorgehensweise bei Restrukturierung:
    - komplexe Verbünde, Selektionen in binäre aufteilen
    - Selektion möglichst „weit unten“ ausführen
    - Selektion und Kreuzprodukt zu Verbund gruppieren
    - aufeinander folgende Selektionen der selben Relation zusammenfassen
    - Projektionen möglichst „weit unten“ ausführen (aber Duplikat-Eliminierung vermeiden)
  - Planoperatoren (können durch *Pipelining* beschleunigt werden):
    - Selektion (*Scan*):
      - \* Kosten:  $C(R)$
      - \* Relationen-Scan (Table-Scan): sequentielles Lesen  
Kosten:  $B(R)$
      - \* Index-Scan: Verwendung eines Index  
Kosten:  $a \cdot \lceil B(R) \cdot \text{Selektivitätsfaktor} \rceil$
    - Projektion: in andere Planoperatoren integriert  
Kosten:  $C(R)$
    - Sortierung
    - Join mit Relationen  $R, S$ :
      - \* Nested-Loop-Join (für *Gleichverbund* mit Index-Zugriff verbesserbar)  
Kosten:  $C(R) + B(R) \cdot C(S)$
      - \* Sorted-Merge-Join (nur für Gleichverbund): sortiere  $R, S$ ; *schrithaltender* Scan  
Kosten:  $C(R) + C(S) + 2 \cdot (B(R) + B(T))$
      - \* Hash-Join (nur für Gleichverbund): kleinere Relation hashen (bei zu großer Relation mehrere Teile); über größere sequentiellen Scan  
Kosten:  $C(R) + C(S)$
    - Duplikat-Eliminierung
    - Gruppierung
  - je nach System/Anwendung Optimierung auf niedrige CPU-/IO-Last
  - *Statistiken* für Wahl des Planoperators sinnvoll (Verteilung der Tupel, Selektivität, ...)



## 10 Recovery

- *Programmfehler*: Absturz des Datenbank-Anwendungsprogramms  $\Rightarrow$  Daten im Puffer und auf Festplatte in undefiniertem Zustand
- *Systemfehler*: DBVS oder BS fällt aus, Hardware-Fehler, ...  $\Rightarrow$  Daten im Puffer verloren, auf Festplatte in undefiniertem Zustand
- *Gerätefehler*: Festplattenausfall  $\Rightarrow$  Daten auf Festplatte sind verloren
- *Transaktionsfehler*: z.B. *Deadlock*, falsche Operationen, Aufruf von `rollback` bzw. `abort`
- *physische Konsistenz*:
  - Korrektheit der Speicherstrukturen, Verweise und Adressen
  - alle Indizes sind vollständig und stimmen mit Primärdaten überein
- *logische Konsistenz*:
  - Korrektheit der Inhalte
  - Referentielle Integrität, Primärschlüsseleigenschaft und eigene Assertions sind erfüllt
  - erfordert physische Konsistenz
- Nach Fehler soll ein logisch konsistenter Zustand erreicht werden:
  - der Zustand vor Beginn der unvollständigen Änderungen durch *Rückgängigmachen* dieser (*undo*)
    - \* *partial*: nach Transaktionsfehler Zurücksetzen der fehlgeschlagenen Transaktion
    - \* *global*: nach Systemfehler mit Verlust des Hauptspeicherinhalts Zurücksetzen aller unvollständigen Transaktion
    - \* Logging-Informationen müssen *vor* dem Einbringen gespeichert werden (*write-ahead log*, WAL)
  - der Zustand nach Abschluss aller Änderungen durch *Vervollständigung* bzw. *Wiederholung* der unvollständigen Änderungen (*redo*)
    - \* *partial*: nach Systemfehler mit Verlust des Hauptspeicherinhalts Wiederholen aller verlorengegangenen Änderungen von abgeschlossenen Transaktionen
    - \* *global*: nach Gerätefehler Einspielen des Backups und Nachvollziehen aller danach erfolgreichen Transaktionen
    - \* Logging-Informationen müssen *vor* dem Melden des erfolgreichen Abschlusses geschrieben werden
- *Sicherung* und *Protokollierung (Logging)* immer notwendig
- auch Einbringstrategie von Bedeutung:
  - „Wann *darf* geänderte Seite auf die Festplatte geschrieben werden?“
    - \* *Steal*: auch schon vor Ende der Transaktion bei Verdrängung aus dem Puffer
    - \* *NoSteal*: erst am Ende der erfolgreichen Transaktion
  - „Wann *muss* geänderte Seite auf die Festplatte geschrieben werden?“
    - \* *NoForce*: erst bei Verdrängung aus dem Puffer (also auch nach Ende einer Transaktion)
    - \* *Force*: spätestens am Ende der erfolgreichen Transaktion
  - „Wie werden geänderte Seiten auf die Festplatte geschrieben?“
    - \* *NotAtomic*: direktes Einbringen, in-place
    - \* *Atomic*: indirektes Einbringen, „Umschalten“ von altem auf neuen Zustand

- Protokollverfahren:
  - *physisch*:
    - \* *Zustandprotokollierung*: *before-image* für undo, *after-image* für redo, auf Ebene von Seiten oder Sätzen
    - \* *Seitenprotokollierung*: für jede geänderte Seite *before-image* und *after-image* sichern
    - \* *Eintragsprotokollierung*: nur geänderte Teile einer Seite
- Begrenzung des Recovery-Aufwands durch *Sicherungspunkte (checkpoints)*:
  - *transaction-oriented checkpoint*: Einbringung mittels Force
  - *transaction-consistent checkpoint*: Beginn neuer Transaktionen verhindern, auf Abschluss der laufenden warten, dann sichern
  - *action-consistent checkpoint*: keine *Änderungsoperation* darf aktiv sein, dann sichern; da aber Transaktionen laufen nur Begrenzung von Redo-Recovery
- *Wiederherstellungsprozedur*:
  - Analyse von letztem Checkpoint bis zum Log-Ende
  - erfolgreiche Transaktionen gegebenenfalls wiederholen
  - fehlgeschlagene Transaktionen von „neu nach alt“ rückgängig machen