1 Einführung

Datenabstraktion: auch Datenkapselung, Datenunabhängigkeit, ...
- Speichern und Wiedergewinnen von Daten ohne Kenntnis der Details der Speicherung
- keine Adressen oder Bytefolgen, sondern Namen und Typen
- Persistente (dauerhafte) Speicherung
- Wiedergewinnen = Auffinden und Aushändigen

1.1 Schichtenbildung

Zur Strukturierung komplexer Software-Systeme

Vorteile:
- höhere Ebenen werden einfacher, weil sie Dienste der tieferen Ebenen benutzen können
- Änderungen auf höheren Ebenen haben keinen Einfluss auf tiefere Ebenen
- höhere Ebenen können abgetrennt werden, tiefere sind auch allein funktionstüchtig
- tiefere Ebenen können getestet werden, bevor die höheren Ebenen lauffähig sind

1.2 Datenbanktechnologie

Konzepte, Methoden, Werkzeuge und Systeme für die dauerhafte, zuverlässige, unabhängige Verwaltung und komfortable, flexible Benutzung von großen, integrierten, mehrfach benutzbaren Datenbanken

Einordnung wichtiger DBS-Konzepte:
- Datenmodelle, Schemata, Sichten:
  Datenintegration, anwendungsorientierte Datenbeschreibung, Datenunabhängigkeit
- Transaktionen:
  Konsistenzkontrolle, Mehrbenutzerbetrieb, Datenintegrität, Wiederanlauf
- Datenschutz, Programmierspracheneinbettung:
  Zugriffskontrolle, Programmankopplung
- Leistungsaspekte, Verfügbarkeit:
  Ad-hoc-Anfragen, Zugriffspfade, Speicherungsstrukturen, Verteilung

2 Dateiverwaltung

Basis-Datenverwaltungsdienst des Betriebssystems
Schichtenbildung: physische Speichergeräte, darüber logische Speichergeräte und Dateien
Nichtflüchtig: Datenträger, rein passiv, dauerhafte Speicherung auch ohne Strom
Speichergeräte: Magnetband (linearer Zugriff), Magnetplattenspeicher (wahlfrei)

2.1 Physische Speichergeräte

- Magnetplattenspeicher:
  feste Sruktur (Zylinder, Spuren, Slots) und Größe, wahlfreier Zugriff
- Magnetbandspeicher:
  kostengünstig, sequentieller Zugriff, Sicherung und Archivierung
- alternative Speicher:
  Optische Speicher, MEMS, Flash/SSDs, Holographie, DRAM, NVRAM, ...

2.2 Logische Speichergeräte

Erhöhung der Störsicherheit, Einzelheiten der E/A-Prozeduren verdecken, Satz von E/A-Prozeduren parametrisieren (Gerätetreiber), flexible Zuordnung von logischen Geräten zu den physischen
physische Adressierung der Slots:
- `readBlock (int CylinderNo, int TrackNo, int SlotNo, char *BlockBuffer)`
- `writeBlock (int CylinderNo, int TrackNo, int SlotNo, char *BlockBuffer)`

Vorteile: schnell, maximale Speicherausnutzung, keine Verwaltungsdaten
Nachteile: kein Schutz, Wissen was wo hin gehört nur außerhalb des Systems, Überschreiben eines vermeintlich freien Blocks, ...

2.3 Dateiverwaltung eines Betriebssystems

Verwaltung von Dateien: haben Namen, dynamisch erweiterbar
Sog. blockorientierte Zugriffsmethode für eine Datei:

- `BlockFile (char *FileName, char Mode, int *BlockSize)`:
  öffnet Datei mit angegebenen Namen, richtet Dateikontrollblock ein
  Schließen löscht Dateikontrollblock
- `append (int NumberOfBlocks)`:
  erweitert um angegebene Zahl von Blöcken
- `write (int BlockNo, char *BlockBuffer)`:
  überschreibt angegebenen Block mit bereitgestelltem Inhalt
- `read (int BlockNo, char *BlockBuffer)`
- `size ()`
- `drop (int NumberOfBlocks)`:
  invers zu append

Dateikatalog: Verwaltungsdatenstruktur: liegt selbst auf der Platte, für jede Datei Abbildung des Dateinamens auf eine Folge von Blöcken
Freispeicherverwaltung: liefert die unbenutzten Blöcke auf einer Platte

Vergleich:

Geschwindigkeit	**direkt**	**blockorientiert**
Geschwindigkeit	sehr schnell	langsamer: Katalogzugriff
Verwaltungsdaten	keine	Katalog
Schutz	keiner	Prüfung von Zugriffsrechten beim Öffnen
Erweiterung einer Datei	nur mit externer Absprache	systemverwaltet
Reorganisation (Versch. v. Blöcken)	Programmänderung	Katalogänderung, Programme stabil
Wechsel des Plattenspeichertyps	Programmänderung	Katalogänderung, Programme stabil

Anwendungsprogramme, die blockorientierten Dateizugriff verwenden, sind unabhängig von verwendeten Speichergeräten → Information Hiding

3 Sätze

Block: Einheit des Transports zw. Hintergrund- und Hauptspeicher
Größe von Blöcken: vorgegeben
- kleine Blöcke: viele E/A-Operationen
- große Blöcke: Mitlesen/schreiben oder Platzverschwendung

⇒ neue Abstraktion **Satz** als Schicht oberhalb der Block-Dateien:

- genau die Daten, die zu einem Gegenstand der Anwendung gehören
- Größe von Anwendung bestimmt, variable Länge
- Satz hier nur Folge von Bytes

Satz-Datei: Menge von Sätzen variabler Länge **Blockung**: Sätze in Blöcken.
Feste Satzlänge: Feste Zahl von Sätzen pro Block (aus Blockgröße und Satzlänge)
Variable Satzlänge: Zahl der Sätze ändert sich von Block zu Block, Reihenfolge der Sätze beliebig

3.1 Sequentielle Satzdatei

Folge von Sätzen fester oder variabler Länge
Seq. Lesen, Schreiben und Anhängen, kein wahlfreies Lesen o. Einfügen/Ändern eines Satzes
Zugriffsoperationen:

- Datei öffnen: Name, Modus, Satzlänge
- Nächsten Satz lesen: Länge, Buffer
- Nächsten Satz schreiben: Buffer, Länge
- Schließen der Datei

Blockunabhängigkeit: Blöcke als Einheit des Transfers zw. Speichergerät und Hauptspeicher sind nicht mehr sichtbar, können durch Reorganisation verändert werden
Pufferung: mehr Speicherplatz bereitstellen für bessere Leistung, völlig unsichtbar für Benutzer der Schnittstelle.

3.2 Direktzugriff auf Sätze

Generelle Frage: wo ablegen, wie wiederfinden?
Direkter Zugriff auf einzelne Sätze über ihre Satzadresse
Satzadresse: beim Einfügen eines Satzen vom DBVS zugeteilt, eindeutig, unveränderlich

TID (Tuple IDentifier) - Konzept:
Satzadressierung über Indirektion innerhalb der Blöcke:

- Hilfsstruktur: Array mit Anfangsadressen aller Sätze dieses Blocks
- Satzadresse ist Paar aus Blocknummer und Index in diesem Feld
- Zugriff auf den Satz nur ein Blockzugriff

Löschen eines Satzes:

- Eintrag des Felds als ungültig kennzeichnen
- alle anderen Sätze können verschoben werden, es ändern sich nur ihre Anfangsadressen im Positionsindex

Änderungsoperationen auf einem Satz:

- Satz schrumpft/wird größer
  alle Sätze innerhalb des Blocks verschoben, Positionsindex anpassen
- Satz passt nicht mehr in den Block: Verschieben in einen anderen Block
  in altem Block verbleibt Verweis auf neue Satzadresse, die auf neuen Block verweist

Direkte Satzdateien:
mithilfe der TIDs einzelne Sätze bel. zugreifbar, änderbar und löschbar
Freispeicherverwaltung:
in jedem Block am Anfang Zahl der freien Bytes → aufwändig

4 Schlüsselzugriff

Satzadressen haben nichts mit Anwendungsdaten zu tun. Deshalb: über Inhalt eines Satzes zugreifen können (Suchschlüssel)

4.1 Hashing

Berechnung der Speicherposition (Blocknummer genügt) aus Schlüsselwert
Hash-Funktion: möglichst gleichmäßige Verteilung der Sätze (modulo q)

Überlaufbehandlung:
Ausweichen auf Nachbar-Buckets (Open Addressing):

- + kein zusätzlicher Speicherplatz
- - nicht nur Treffer, sondern auch Überläufer, bei Löschen diese zurückholen
- - in Nachbarbuckets u.U. zusätzliche Überläufe

Overflow-Buckets (mit Separate Chaining):
Primärbucket zeigt auf sein Überlauf-Bucket

- - Speicherplatzbedarf
- + keine Mischung von Sätzen, keine Beeinträchtigung der Nachbarbuckets

Hashfunktion nur intern → wählt Systemadministrator
Bewertung: schneller Zugriff über Schlüssel, Sätze durcheinandergewürfelt, Speicherplatz im Vorhinein belgen (zu groß: Verschwendung, zu klein: Reorganisation), Hashing nur nach einem Schlüssel

4.2 Virtuelles Hashing

mit q Buckets beginnen: jeder mit maximal b Sätzen $\Rightarrow q \times b$ Sätze. Belegungsfaktor β.
Wenn β größer als Schwellwert, Menge der Buckets vergrößern
VH1: auf einen Schlag q, 2q, 4q, .. neue Blöcke belegen, direkt hinter den bisherigen Buckets. Umspeichern Stück für Stück
Lineares Hashing: einen neuen Bucket hinten anfügen; Bucket, auf dem Positionszeiger p steht, aufteilen, Positionszeiger für aufgeteilte Blöcke. Dazu Hashfunktion für erweiterte Buckets anpassen

4.3 B-Bäume

n Einträge pro Knoten, $k \leq n \leq 2k$
Eintrag: (K_i, D_i, P_i) mit K_i Schlüsselwert, D_i Datensatz, P_i Zeiger auf Blocknummer des Nachfolgeknoten
Einträge nach Schlüsselwert aufsteigend sortiert
Bedeutung:

- alle Schlüsselwerte im Unterbaum, auf den P_0 zeigt, $\leq K_1$
- alle Schlüsselwerte im Unterbaum, auf den P_i zeigt $(0 < i < n)$, $> K_i$ und $\leq K_{i+1}$
- alle Schlüsselwerte im Unterbaum von P_n größer als K_n

B-Baum-Parameter:

- k: errechnet sich aus Blockgröße, max. $2k$ Einträge pro Block
- h: Höhe des Baums (Anzahl Kanten von Wurzel bis Blatt + 1, also Anzahl Ebenen)
  ergibt sich aus Anzahl der gespeicherten Datenelemente und Einfügereihenfolge

Eigenschaften:

- Jeder Pfad hat diesselbe Länge $h - 1$
- Jeder Knoten (außer Wurzel und Blattknoten) hat mind. $k + 1$ Nachfolger
- Wurzelknoten ist entweder Blattknoten oder hat mind. 2 Nachfolger
- Jeder Knoten hat höchstens $2k + 1$ Nachfolger
- Jeder Knoten bis auf die Wurzel ist immer mindestens halb voll

Suche: beginnend mit Wurzelknoten, Knoten von links nach rechts durchsuchen:

- K_i gesuchter Schlüsselwert? $\Rightarrow$ Satz gefunden
- $K_i >$ ges. Wert $\Rightarrow$ Suche in Wurzel des an P_{i-1} hängenden Unterbaums ‘
- $K_i <$ ges. Wert $\Rightarrow$ wiederhole Vergleich mit K_{i+1}
- $K_n <$ ges. Wert $\Rightarrow$ Suche im Unterbaum von P_n

Einfügen: nur in Blattknoten, d.h. erst Abstieg, im gefundenen Blattknoten einfügen
Wenn Blattknoten schon voll:

- Splitt: $2k + 1$ Sätze halbe halbe aufteilen
- die ersten k Sätze in den linken Block
- die zweiten k Sätze in den rechten Block
- den mittleren $k + 1$-ten Satz als Diskriminator in die Stufe höher einfügen
- Falls übergeordneter Knoten voll: Splitt wiederholen
- Splitt des Wurzelknotens: Erzeugung von 2 neuen Knoten:
  einen neuen Wurzelknoten und einen neuen Teil
  dann (und nur dann) wächst die Höhe des Baumes um 1

Löschen:

- Suche Knoten für zu löschenden Schlüssel S
- falls S Blattknoten, lösche S und behandle ggf. entstehenden Unterlauf
- falls S im inneren Knoten, dann untersuch linken und rechten Unterbaum von S:
  - Betrachte Blattknoten mit direktem Vorgänger S' und Blattknoten mit direktem Nachfolger S''
  - wähl den aus, der mehr Elemente hat (sonst zufällig)
  - Ersetze S durch S' bzw. S''
  - lösch S' bzw. S'' im Blattknoten und behandle ggf. Unterlauf

Variante: B*-Baum
Alle Sätze werden in Blattknoten abgelegt, innere Knoten enthalten nur Verzweigungsinformation
Löschen im B*-Baum:

- suche zu löschenden Eintrag im Baum
- Entsteht durch das Löschen ein Unterlauf?
  nein: entferne Satz
  ja
  - prüf Blatt zusammen mit Nachbarknoten
  - ist die Summe der Einträge in beiden Knoten größer als $2k$
  - NEIN: misch beide Blätter, bei Unterlauf im Vaterknoten: misch innere Knoten analog
  - JA: teile Sätze neu auf beide Knoten auf, so dass ein Knoten jew. die Hälfte der Sätze aufnimmt

Vergleich B- und B*-Baum:

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

4.4 Bitmap-Index

B-Bäume (und Hashing) sinnvoll für Suchschlüssel mit hoher "Selektivität"
Viele gleiche Werte (Geschlecht) $\Rightarrow$ Bitmap

- für jeden Schlüsselwert eine Bitliste
- jedem Satz der Datei ist ein Bit in der Bitliste zugeordnet (feste Reihenfolge)
- Bitwert 1: Schlüssel hat den Wert, zu dem die Liste gehört, 0: hat anderen Wert
- Indexgröße: (Anzahl der Werte) $\times$ (Anzahl der Sätze) Bits

Eigenschaften:

- wächst mit Anzahl möglicher Werte
- besonders interessant bis zu ca. 500 versch. Werten
- Hauptvorteil: einfache und effiziente logische Verknüpfbarkeit

4.5 Primär- und Sekundär-Organisation

Primär-Organisation:

- bestimmt Speicherung der Sätze selbst
- kann sequentiell, direkt oder über Schlüssel sein

Sekundär-Organisation:

- verweist nur auf die Sätze, die nach beliebigen anderen Kriterien abgespeichert wurden
- nur möglich, wenn Primärorganisation Direktzugriff auf einzelnen Satz unterstützt (Satzverweis)
- B(*)-Baum als Sekundär-Organisation (an Stelle des Satzes nur Satzverweis)

4.5.1 Organisation einer Menge von Sätzen

- eine Datei für die Sätze selbst
- für jede Sekundär-Organisation eine eigene Datei
- Einfügen:
  - in die Datei der Sätze (liefert Satzverweis)
  - in jede Index-Datei: Paar (Schlüsselwert, Satzverweis)
- Suchen:
  - read in der Datei der Sätze
  - über Index:
    read im Index, liefert Satzverweise
    in Schleife über alle Treffer: read in der Satzdatei
  - über nicht "indexierte" Felder: mit Schleife in Datei der Sätze

4.5.2 Eindeutige Schlüssel

Primärschlüssel: Schlüssel, bei dem jeder Wert in höchstens einem Satz vorkommen darf
Sekundärschlüssel: nicht eindeutig, darf in mehreren Sätzen gleich sein
Weitere Möglichkeiten: Domänen-orientierte Indexstruktur, Mehrdimensionale Indexstrukturen (R-Bäume)

5 Pufferverwaltung

Im Hauptspeicher Platz für n Blöcke ($n > 1$)

5.1 Blockzugriffe

Zugriff auf Block i (sog. logischer Zugriff), zwei Fälle:

- Block im Puffer
- Block muss von Platte eingelesen werden (Einlagern eines Blocks)
- Einlagern verdrängt einen anderen Block
  wenn geändert worden: muss auf Platte zurückgeschrieben werden
  sonst direkt überschreiben

5.2 Ersetzungsverfahren

Ziel: Minimierung der Zahl physischer Zugriffe bei gegebener Zahl logischer Zugriffe
Ersetzungsstrategie: nach Alter und Benutzungshäufigkeit eines Blocks

- First in, first out (FIFO):
  nur Alter, ungünstig wenn häufig benutzte Blöcke im Speicher bleiben sollen
- Least frequently used (LFU):
  nur Häufigkeit, ungünstig wenn Block Zugriffe angesammelt hat und dann nicht mehr verwendet wird
- Least recently used (LRU):
  bewertet Alter seit letztem Zugriff, aufwändig
- CLOCK (Second Chance):
  LRU-Verhalten mit einfacher Implementierung
  Benutztbit wird bei Zugriff auf 1 gesetzt
  Verdrängung: wenn 1 dann auf 0, wenn 0 wird Block ersetzt

5.3 Schnittstelle einer Pufferverwaltung

Einkapselung der Pufferverwaltung:

- `char *Buffer::fix(BlockFile File, int BlockNo, char Mode)`:
  stellt Block im Puffer zur Verfügung und liefert Anfangsadresse
- `void Buffer::unfix(char *BufferAddress)`:
  gibt Block im Puffer zur Ersetzung frei

5.4 Fehlersituationen

Betriebssystem-Absturz, Hardware-Fehler, Stromausfall → Pufferverwaltung sollte Unterstützung bieten

5.5 Einbringstrategien

Was tun nach Absturz/Hardwarefehler/Stromausfall? (ja, weinen)
Einbringen = Ablegen auf nicht-flüchtigem Speicher derart, dass es nach einem Ausfall verwendet werden kann, kann auch verzögert stattfinden
Segment: linearer, logischer, potenziell unendlicher, aber endlicher Adressraum mit sichtbaren Seitengrenzen

Begriffliche Trennung:

- Block: Block auf physischer Festplatte, nur ganze Blöcke gelesen/ geschrieben
- Kachel: Platz für Block im Datenbankpuffer
- Seite: Addressierung an Pufferschnittstelle, Seite wird auf Blöcke abgebildet, Block im Puffer

Seitenzuordnung: Welche Blöcke für eine Seite?

- **direkt**:
  aufeinander folgende Seiten werden auf aufeinander folgende Blöcke einer Datei abgebildet
  man muss sich nur erste Blocknummer für Segment merken
  im Puffer können Seiten trotzdem verstreut liegen!
- **indirekt**:
  benötigt Hilfsstruktur mit Blocknummer zu jeder Seite
- direkt ist schneller, aber auch unflexibler

Seiteneinbringung:
- **direkt**: wenn auf Platte geschrieben (verdrängt), sofort in Datenbestand eingebracht
- **indirekt**: geänderte Seiten werden nicht sofort bei Verdrängung aus Puffer auf Platte geschrieben
  zu klären: wann und wie wird man die alten Blöcke los?

Schattenspeicher:
- Inhalte aller Seiten eines Segments werden in einem Sicherungsintervall Δt in einem konsistenten Zustand unverändert gehalten
- Sicherunspunkt besteht aus: belegten Seiten, Seitentabelle, Bitliste auf stabilem Speicher
- im Fehlerfall segmentorientiertes Zurückgehen auf letzten Sicherungspunkt

Twin Slots: pro Block zwei Speicherplätze (mit Versionsnummer)

6 Programmschnittstelle

6.1 Relationale Datenbanken

Bisher erreicht: Interne Satzschnittstelle: Benutzer arbeitet mit Segment (Menge von Sätzen) und Satz (variabel lange Folge von Bytes, evtl. mit Schlüssel)
Nächster Schritt: Unabhängigkeit der Programme von Organisationstypen und Indexen, nur noch Satzmengen
Weitere Abstraktion:
- von Dateien zu Relationen (oder Klassen)
- von Sätzen zu Tupeln (oder Objekten)
- von Feldern (Schlüsseln) zu Attributen

6.2 Eingebettetes SQL

Erweiterung der Programmiersprache: Vorübersetzer oder Extra-Compiler
- Bequemer für Programmierer (kompakter)
- Typüberprüfung und Optimierung schon zur Übersetzungszeit → nur einmal

6.3 Unterprogrammaufruf

Unterprogrammschnittstelle (Call-level interface, CLI) eines DBS
Programmiersprache (auch "Wirtssprache" – host language – genannt) bleibt unverändert
- Parameter für die Unterprogramme:
  alle Teile der SQL-Anweisungen einzeln
  oder SQL-Anweisung insgesamt als Zeichenkette – umständlich!
- Typüberprüfung und Optimierung erst zur Ausführungszeit → immer wieder

6.4 O/R-Mapping

Besonderheit für objektorientierte Programmiersprachen.
Objekte durch Werkzeuge in Tupeln abspeichern, Konfiguration mit XML-Dateien

7 Transaktionen

Umgang mit Fehlern und Ausfällen:
- **Programmfehler**: Division durch Null, unzulässige Adressierung, ...
  Ergebnis: Hauptspeicherinhalte verschwunden
  Maßnahme: Daten im Puffer wegwerfen. Platte von Hand bereinigen
- **Systemfehler**: DBVS oder BS oder Strom fällt aus
  Ergebnis: Pufferinhalte verloren, Platte in unbekanntem Zustand
  Maßnahmen: alle Programme, die liefen, von Hand nachvollziehen
- **Gerätefehler**: nichts mehr lesbar, Diebstahl
  Ergebnis: Programme, DBVS und BS können nicht mehr weitermachen
  Maßnahmen: neue Platte, Backup einspielen, alle Änderungen wiederholen

Erwünschte Zustände der Daten auf Platte:

- **Physische Konsistenz**:
  Korrektheit der Speicherungsstrukturen
  Alle Verweise und Adressen (TIDs) stimmen
  Alle Indexe sind vollständig und stimmen mit Primärdaten überein
- **Logische Konsistenz**:
  Korrektheit der Dateninhalte
  alle Bedingungen des Datenmodells und alle benutzerdef. Bedingungen sind erfüllt

Annahmen:

- alle vollständig ausgeführten Datenbank-Operationen hinterlassen einen physisch konsistenten Zustand: das DBVS ist nicht fehlerhaft
- alle vollständig ausgeführten Anwendungsprogramme hinterlassen einen logisch konsistenten Zustand: der Programmierer hat alles richtig gemacht
- Nach einem Fehler: Daten i. Allg. weder physisch noch logisch konsistent

Ziel:

- Systemunterstützung zur Wiederherstellung, am besten automatisch
- Rückgängigmachen der bereits ausgeführten Änderungen (Backward Recovery)
- Wiederholen verlorengegangener Änderungen (Forward Recovery)

Hierzu: geeignete Sicherungs- und Protokollierungsmaßnahmen im laufenden Betrieb

7.1 Transaktion

Transaktion: Folge von DB-Operationen, die, von einem logisch konsistenten Zustand ausgehend, die Datenbank wieder in einen logisch konsistenten Zustand überführt

Bei Fehler...

- vor dem Ende der Transaktion: Alle Änderungen werden automatisch so rückgängig gemacht, dass sich die Daten wieder in dem Zustand befinden, in dem sie zu Beginn der Transaktion waren (Systemgarantie). Anwender kann selbst die TA neu starten.
- nach dem Ende der Transaktion: Alle Ergebnisse der Transaktion, die durch den Fehler verlorengegangen waren, werden automatisch wiederhergestellt

Aufgabe des Anwenders: Transaktion definieren

- DB-Aufruf `commit`:
  Anwendungsprogramm teilt dem DBVS mit, dass es nach einer Reihe von DB-Änderungen wieder einen logisch konsistenten Zustand hergestellt hat oder
  dass es nach einer Reihe von lesenden Zugriffen so weit fertig ist, dass andere die gelesenen Daten auch wieder ändern dürfen
- DB-Aufruf `abort` oder `rollback`:
  Rückkehr in den Anfangszustand auf Wunsch des Anwenders

Charakterisierung:

- Datenbanktransaktion stellt logische Arbeitseinheit dar
- Zusammenfassung von aufeinander folgenden DB-Operationen, die eine Datenbank von einem konsistenten Zustand in einen neuen konsistenten Zustand überführen
- Innerhalb einer Transaktion: vorübergehend logisch inkonsistente Zustände
- DB-Zugriffe nur noch innerhalb von Transaktionen erlaubt
- TAs so kurz wie möglich und so lang wie nötig
- Eine Transaktion ist **atomar**:
  - unteilbar: alle zu einer Transaktion gehörenden Operationen bilden eine Einheit
  - ununterbrechbar: ganz oder gar nicht ausgeführt
- Der Transaktionsmanager garantiert entweder vollständige Ausführung einer Transaktion oder Wirkungslosigkeit der gesamten Transaktion
- **Konsistenzerhaltung**: vor Start sowie nach erfolgreichem Abschluss konsistenter Zustand
- **Isolation**: Wenn mehrere Transaktionen gleichzeitig auf demselben Datenbestand laufen, laufen diese isoliert voneinander ab. D.h. solange eine Transaktion läuft, darf keine andere Transaktion von ihr bereits durchgeführte Änderungen lesen und benutzen (unvollständig/inkonsistent, Transaktion kann scheitern).

Eine Transaktion T1 darf entweder nur den Zustand vor Ausführung einer anderen Transaktion T2 sehen oder nur den Zustand nach Ausführung von T2.
Die Synchronisation muss so erfolgen, dass die Wirkung auf den Datenbestand diesselbe ist wie bei einer strikt seriellen Ausführung. Blockierung möglichst kurz machen.

- **Dauerhaftigkeit**: eine als erfolgreich abgeschlossen gemeldete TA ist persistent

7.2 Kurzgesagt...

ACID-Eigenschaften von Transaktionen:

- Atomarität: Unteilbarkeit, Alles-oder-Nichts-Prinzip
- Konsistenz: alle Integritätsbedingungen wurden eingehalten
- Isolation: keine Zwischenergebnisse von anderen Transaktionen
- Dauerhaftigkeit: alle Ergebnisse erfolgreicher Transaktionen müssen persistent gemacht worden sein, bevor Erfolg an Anwendung gemeldet werden darf

8 Speicherung von Tupeln und Relationen

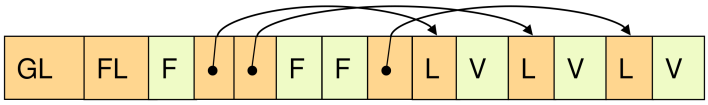
8.1 Speicherung von Tupeln in Sätzen

Sätze sind aus einzelnen Feldern zusammengesetzt.
Satztyp: Menge von Sätzen mit gleicher Struktur. Jedem Satz wird beim Abspeichern ein Satztyp zugeordnet.
Annahmen: variable Satzlänge, ein Satz vollständig in einer Seite ablegbar, Reihenfolge spielt keine Rolle.

Anforderungen an Tupelspeicherung:

- Speicherplatzeffizienz
  Variable Länge, undef. Werte gar nicht speichern, mögl. wenig Hilfsstrukturen
- Direkter Zugriff auf Felder
  ohne andere Felder zu lesen, direkt zur Anfangs-Byte-Position des Feldes in einem Satz
- Flexibilität
  Hinzufügen von neuen Feldern bei allen Sätzen
  Löschen eines Feldes aus allen Sätzen

Speicherungsstruktur in Sätzen: Eingebettete Längenfelder mit Zeigern



- fester Strukturteil: Felder fester Länge und Zeiger
- Variabel lange Felder ans Ende des festen Stukturteils legen
- Satzinterne Adresse aus Katalogdaten berechenbar

8.2 Speicherung von Relationen, C-Store

Spaltenweise abspeichern ("Column-Store"): auf Lesen hin optimiert, nicht auf Schreiben → **C-Store**
Speichert Sammlung von Spaltengruppen (Gruppe von Spalten = Projektion) mit

1. Schreibspeicher (WS): für schnelles Einfügen und Ändern von Tupeln
2. Lese-optimierter Speicher (RS) für umfangreiche Analysen
3. Tuple Mover dazwischen, im Hintergrund
4. Änderungen durch Löschen und Einfügen realisieren

- Projektion: ein oder mehrere Attribute einer Tabelle, Duplikate bleiben erhalten
  Spaltenweise Speicherung der Projektionen, jedes Attribut einzeln (Sortierschlüssel: eines der Atttribute)
- für jede Tabelle: überdeckende Menge von Projektionen
  Rekonstruktion vollständiger Tupel muss möglich sein
- Speicherschlüssel (Storage Keys):
  in jedem Segment mit jedem Attributwert verbunden
  gleicher SK bei versch. Attributen: gleiches Tupel

- Verbund-Indexe:
  T1 und T2 Projektionen der Tabelle T, Join Index von M Segm. in T1 zu N Segm. in T2:
  Sammlung von M Tabellen, je eine pro Segment von T1, mit Tupeln:
  `s: SID in T2, k: Storage Key in Segment s`

8.2.1 physische Strukturen

Spalten: Komprimierung...
- wenig verschiedene Werte...
  - sortiert:
    Tripel (v, f, n): Value, first position, Anzahl gleicher Werte
    organisiert in B-Baum mit dichter Packung und großen Seiten
  - unsortiert:
    Paare (v, b): Wert, Bitmap
    B-Baum für Abbildung von Positionen in einer Spalte auf die Werte in dieser Spalte
- viele verschiedene Werte...
  - sortiert:
    Delta-Codierung: Differenzen vom Vorgänger-Wert speichern
    in jeder Seite zuerst absoluten Wert (und Speicherschlüssel)
    B-Baum mit dichter Packung
  - unsortiert:
    unkomprimiert, B-Baum als Sekundär-Organisation mögl.
- bei Zeichenketten: Wörterbuch
  Sortierte Liste aller vorkommender Werte, in der Spalte nur noch Indexposition in dieser Liste

Verbund-Indexe: Segment-ID, Speicherschlüssel, Speicherung wie die anderen Attribute auch

8.2.2 Schreibspeicher

- keine zwei versch. Optimierer schreiben:
  WS und RS haben gleiche Projektionen und Verbund-Indexe
- Speicherschlüssel: explizit gespeichert in jeder Projektion und jedem Segment
- gleiche Segmentierung wie RS, keine Komprimierung

8.2.3 weitere Aufgaben

- Speicherverwaltung: v.a. Allokation
- Änderungen und Transaktionen
- Tuple Mover: von WS in RS (Hintergrundaufgabe)
- Anfrageausführung

9 Anfrageverarbeitung

9.1 Einführung

Realisierung eines mengenorientierten Zugriffs, nicht nur auf einzelne Sätze
Abbildung von mengenorientierten Operatoren auf satzorientierte Operatoren und Benutzung von Indexstrukturen

Verarbeitung:
- Überprüfung auf syntaktische Korrektheit
- Überprüfung von Zugriffsberechtigung und Integritätsbedingungen
- Anfrageoptimierung
- Ausführung

Zentrale Aufgabe für RDBVS: Umsetzung deskriptiver Anfragen in eine optimale Folge interner DBS-Operationen (an der Satzschnittstelle)
Probleme: hohe Komplexität, zusätzliche Anforderungen, was ist optimal (max. Durchsatz, min. Antwortzeit, Einhalten von Zeitschranken)

9.2 Abstrakte Sicht auf die Anfrageverarbeitung

Aufteilung der Anfrageverarbeitung (Query Processing):
- Anfrageverarbeitung (AV): liefert Anfrageausführungsplan zur Übersetzungszeit
- Anfrageausführung (AA): physischer DB-Prozessor, tatsächliche Ausführung zur Laufzeit

Phasen der Anfrageverarbeitung:
- **lexikalische und sytaktische Analyse**:
  korrekte Syntax, Erstellen eines Anfragebaums
- **semantische Analyse**:
  Feststelllen der Existenz und Gültigkeit der referenzierten Relationen und Attribute, Ersetzen der externen durch interne Namen, Konvertierung der Werte vom externen Format in interne Darstellung
- **Zugriffs- und Integritätskontrolle**:
  Durchführung einfacher Integritätskontrollen, Generierung von Laufzeitaktionen für werteabhängige Kontrollen
- **Standardisierung und Vereinfachung**:
  Überführung des Anfragebaums in eine Normalform, Elimination von Redundanzen
- **Restrukturierung und Transformation**:
  Algebraische Verbesserung (Anwendung von heuristische Regeln)
  Nicht-algebraische Verbesserung (Ersetzen und ggf. Zusammenfassen der log. Operatoren durch Planoperatoren)
  Auswahl der günstigsten Planalternative
- **Code-Generierung**:
  Generierung eines zugeschnittenen Programms für die vorgegebene Anfrage, Erzeugung eines ausführbaren Zugriffsmoduls

9.3 Interndarstellung einer Anfrage

Wie wird eine SQL-Anfrage intern repräsentiert?
Relationale Algebra definiert relationale logische Operatoren, die für die interne Darstellung einer Anfrage in Form eines Anfragebaums geeignet sind

Mengenorientierte Operationen:
- SEL(R, pred(...)): Auswahl einer Teilmenge (Zeilen)
- PROJ(R,L): Auswahl aller Tupel bzgl. einer Teilmenge L von Attributen (Spalten)
- CROSS(R,S): Kreuzprodukt zweier Relationen
- JOIN(R,S,pred): Verbinden zweier Relationen gemäß eines Prädikats über Attributen aus beiden Relationen
- UNION(R,S)
- INTERSECT(R,S)
- EXCEPT(R,S)
- RENAME(R,[Rnew,]((A1, A1new), (A2,A2new), ...)): Umbenennung
- DUP-ELIM(R)
- SUM(R,attr), AVG(R,attr), MIN(R,attr), MAX(R,attr): numerisch
- COUNT(R)
- GROUP(R,L,agg): Gruppierungsattribute L und Aggregationen agg
- G-PROJ(R,L): L liste von Ausdrücken zur Berechnung von neuen Attributwerten
- SORT(R,L) mit L Liste der Attribute, nach denen sortiert wird, Ergebnis ist Liste!
- OUTER-JOIN(R,S,pred,case): left, right oder full outer join

Operatorbaum: effiziente Datenstruktur mit geeigneten Zugriffsfunktionen
- prozedurale Darstellung einer deskriptiven, mengenorientierten Anfrage
- Knoten sind Operatoren der rel. Algebra
- Blattknoten sind Relationen
- gerichtete Kanten repräsentieren den Datenfluss

9.4 Standardisierung und Vereinfachung

Standardisierung: Wahl einer Normalform
Vereinfachung: Idempotenzregeln, Ausdrücke mit leeren Relationen

9.5 Restrukturierung

Äquivalente Umformung des Operatorbaums

1. n-facher Verbund kann durch Folge von binären Verbunden ersetzt werden und umgekehrt
2. Verbund ist kommutativ
3. Verbund ist assoziativ
4. Selektionen können zusammengefasst werden:
   `SEL(SEL(R,pred1),pred2) = SEL(R,(pred1 AND pred2))`
5. Projektionen können zusammengefasst werden:
   `PROJ(PROJ(R,L1),L2) = PROJ(R,L2)`
6. Selektion und Verbund dürfen vertauscht werden:
   `SEL(JOIN(R,S,pred1),pred2(R)) = JOIN(SEL(R,pred2),S,pred1)`
7. Selektion darf mit Vereinigung und Differenz vertauscht werden
   `SEL(UNION(R,S),pred) = UNION(SEL(R,pred),SEL(S,pred))`
8. Selektion und Kreuzprodukt können zu Verbund zusammengefasst werden:
   `SEL(CROSS(R,S),pred) = JOIN(R,S,pred)`
9. Selektionen mit mehreren Prädikat-Termen separieren in Selektionen mit jeweils einem Prädikat-Term

Ziel: Zwischenergebnisse möglichst klein halten.
Vereinfacht:

- Komplexe Verbundoperationen zerlegen in binäre Verbunde
- Selektionen mit mehreren Prädikat-Termen separieren in Selektionen mit jeweils einem Prädikat-Term
- Selektionen so früh wie möglich ausführen, d.h. Selektionen hinunterschieben zu den Blättern des Anfragebaums
- Selektionen und Kreuzprodukt zu Verbund zusammenfassen, wenn das Selektionsprädikat Attribute aus den beiden Relationen verwendet
- Einfache Selektionen wieder zusammenfassen, d.h. aufeinanderfolgende Selektionen (derselben Relation) gruppieren
- Projektionen so früh wie möglich ausführen, d.h. Projektionen hinunterschieben zu den Blättern des Anfragebaums, dabei aber die teure Duplikat-Eliminierung vermeiden!

10 Relationale Operatoren

10.1 Transformation

Aufgabe: Ersetzen der Logischen Operatoren durch ausführbare Planoperatoren

Teilprobleme:

- Gruppierung von direkt benachbarten Operatoren zur Auswertung durch einen einzelnen Planoperator
- Bestimmung der Verknüpfungsreihenfolge bei Verbundoperationen
- Erkennen gemeinsamer Teilbäume (einmalige Berechnung)

Relationale Planoperatoren:

- Ausführbar (Komponente des DBMS)
- Parameter (Eingabe-Relationen, Indexstrukturen, Bedingungen)
- Voraussetzungen (Vorhandensein best. Speicherungsstrukturen)
- Ergebnis (ganze Relation, nächstes Tupel, nächste n Tupel)
- Kosten der Verarbeitung (Zeit, Speicher, CPU, ...)

SQL erlaubt komplexe Anfragen über k Relationen

Planoperatoren:

- **Selektion**: mehrere Planoperatoren zur Auswahl
  nutzt Scan-Operator: Start- und Stopp-Bedingung
  Relationen-Scan: seq. Lesen aller Tupel einer Relation
  Index-Scan: direkt auf erstes passendes Tupel, dann ggf. seq. die anderen
- **Projektion**: typischerweise mit im Planoperator von Sortierung, Selektion oder Join durchgeführt
- **Sortierung**
- **Join** über mehreren Relationen: Zerlegung in $n-1$ Zwei-Wege-Verbunde

Join ist lahm und häufig verwendet → Optimieren!

- **Nested-Loop**: für jedes Element der äußeren Tabelle die innere Tabelle ablaufen
  mit Indexzugriff: anstatt innere Tabelle mit Indexzugriff auf Objekte zugreifen
- **Sort-Merge**: Sortierung mit Eliminierung, dann schritthaltende Scans
- **Hash-Verbund**: Partitionierung der inneren Tabelle. Laden der Partitionen in eine Hash-Tabelle. Probing der äußeren Tabelle mit HT.
- **Duplikat-Eliminierung**: Sortierung oder Hashing
- **Grupperung**: Sortierung und Scan mit Aggregation pro Gruppe oder...
  Hashing: Abbildung auf Zähler, bisherige Summe, bisheriges Min/Max

10.2 Anfrageoptimierung

Ziel: Maximierung des Outputs bei geg. Ressourcen oder Minimierung der Ressourcennutzung für geg. Output

Kostenarten:

- Berechnungskosten (CPU-Kosten, Pfadlängen)
- E/A-Kosten (Anz. phys. Referenzen)
- Speicherungskosten
- Kommunikationskosten

Erstellung und Auswahl von Ausführungsplänen:
Eingabe: optimierter Anfragebaum, Speicherungsstrukturen, Kostenmodell
Ausgabe: möglichst guter Ausführungsplan
Annahme: alle Datenelemente und Attribute sind gleichverteilt
Vorgehensweise:

- Generieren aller vernünftigen logischen Ausführungspläne zur Auswertung der vorliegenden Anfrage
- Vervollständigen der Ausführungspläne mit Einzelheiten aus der physischen Datenrepräsentation
- Bewertung der generierten Alternativen und Auswahl des billigsten Ausführungsplans gemäß Kostenmodell

Alternative Ausführungspläne: versch. Implementierungen von Planoperatoren, Reihenfolge verändern
Sehr große Suchräume bei komplexen Anfragen mit allen Alternativen

Suchstrategien für einen guten Ausführungsplan:
Auffinden eines guten Plan mit mögl. kleiner Anzahl generierter Pläne
Strategieklassen:

- voll-enumerativ
- beschränkt-enumerativ
- zufallsgesteuert

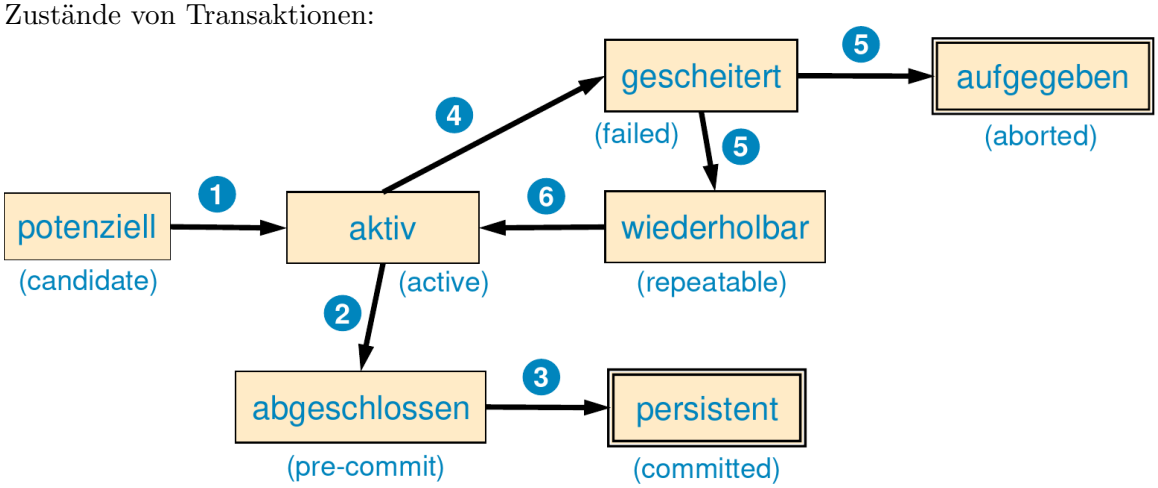
Berechnung der Zugriffskosten:
Kostenformel:

- Gewichtetes Maß für E/A- und CPU-Belastung:
  Kosten = #phys. Seitenzugriffe + W · #Aufrufe d. Zugriffssystems
  W ist Verhältnis des Aufwands für einen Aufruf des Zugriffssystems zum Aufwand für einen Seitenzugriff
- Gewichtung? Minimierung abhängig vom System:
  E/A > CPU: W_{cpu} = #Instr-Zugriffsystem / #Instr-E/A
  E/A < CPU: W_{io} = #Instr-Zugriffssystem / (#Instr-E/A + Zugriffszeit × MIPS-Rate)

Selektivitätsabschätzung: erwarteter Anteil an Tupeln, die ein Prädikat erfüllen
größtenteils random...

11 Synchronisation

Zustände von Transaktionen:



1. Inkarnieren: TA ist angemeldet, wechselt zu aktiv
2. Beenden: TA ist beendet, Änderungen noch nicht permanent eingebracht
3. Festschreiben: Änderungen werden eingebracht
4. Abbrechen: TA ist fehlgeschlagen, noch nicht zurückgesetzt
5. Zurücksetzen: Änderungen werden rückgängig gemacht
6. Neustarten: TA wird wiederholt

11.1 Synchronisation allgemein

Ziel: Erhaltung der Tranaktionskonsistenz im Mehrbenutzerbetrieb
Gründe für Mehrbenutzerbetrieb:

- Verteilung der Clients generell
- Kommunikationsvorgänge in verteilten Systemen
- Prozessornutzung bei system-/benutzungsbedingter Transaktionsunterbrechungen (Wartezeiten wg. I/O)

Gegenstand der Synchronisation: Vermeidung der ggseitigen Beeinflussung von Lese- und Schreiboperationen

Vorgehensweise:

- Anomalien bei fehlender Synchronisation:
  lost update, dirty write, dirty read, non-repeatable read
- Unbefriedigend: Serialisierung
- Lösung: "Logischer" Einbenutzerbetrieb
- exklusives Änderungsrecht für Schreiber, Leser muss Daten schützen

11.2 Serialisierbarkeit

Problem: nicht alle verzahnten Abläufe sind korrekt

Serialisierbarer Ablauf: Ablauf von Transaktionen ist zu irgendeinem seriellen Ablauf der in ihm enthaltenen Transaktionen äquivalent ist. Äquivalent heißt, dass bei solchen Operations-Paaren die Reihenfolge in beiden Abläufen gleich sein muss.

Nachweis der Serialisierbarkeit: Abhängigkeitsgraph:
Knoten: einzelne Transaktionen
Kanten: Abhängigkeiten zw. Transaktionen
Serialisierbarkeit, wenn der Abhängigkeitsgraph keine Zyklen enthält

→ zwei Abläufe sind äquivalent, wenn sie diesselben erfolgreich abgeschlossenen Transaktionen enthalten und ihre Abhängigkeitsgraphen gleich sind
Verhindert werden müssen:

- dirty write: $w_1[x]...w_2[x]$
- dirty read: $w_1[x]...r_2[x]$
- non-repeatable read: $r_1[x]...w_2[x]$
- phantom: $r_1[P]...w_2[yinP]$

11.3 Implementierungsmethoden

Einführung von Sperren für Zugriffe auf Datenobjekte

Umgang mit Sperren:
- jedes Datenobjekt, auf das zugegriffen werden soll, muss vorher gesperrt werden
- Transaktion fordert eine Sperre, die sie bereits besitzt, nicht nochmal an
- Transaktion muss die von anderen Transaktionen gesetzten Sperren beachten
- am Ende einer Transaktion sind alle Sperren wieder freizugeben

Wann Sperren erwerben?
- Statisch: zu Beginn der Transaktion alle Sperren anfordern
- Dynamisch: bei Bedarf anfordern → Deadlocks

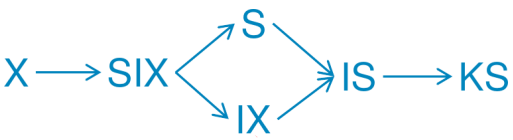
Wann freigeben? Bis zum Ende der Transaktion - oder falls nicht mehr benötigt

Probleme mit Sperren:
- Sperrgranulat Tupel: Transaktionen brauchen sehr viele Tupel einer Relation
- Phantom-Problem: Sperren nur auf existierende Tupel
  ⇒ Hierarchische Schachtelung der Datenobjekte

Arten von Sperren:
- X (eXclusive)-Sperre: Schreibsperre
- S (Shared)-Sperre: Lesesperre
- IS-Sperre (intention share):
  auf untergeordnete Objekte wird nur lesend zugegriffen
- IX-Sperre (intention exclusive):
  auf untergeordnete Objekte wird auch schreibend zugegriffen
- SIX = S + IX (share and intention exclusive):
  Sperrt Objekt in S-Modus, auf tieferen Ebenen nur noch IX-/X-Sperren für zu ändernde Objekte
  z.B. alle Tupel müssen gelesen werden, nur einige davon geändert
- Nutzung einer Untermenge wird angezeigt
- Knoten mit S oder IS sperren → alle Vorgänger im IX- oder IS-Modus
  Knoten mit X oder IX sperren → alle Vorgänger im IX-Modus
- Knoten freigeben: Bottom Up

Kompatibilitätsmatrix:

Objekt → ↓ Transaktion	IS	IX	S	SIX	X
IS	+	+	+	+	-
IX	+	+	-	-	-
S	+	-	+	-	-
SIX	+	-	-	-	-
X	-	-	-	-	-



11.4 Deadlocks (Verklemmungen)

Voraussetzungen:
- gleichzeitiger Zugriff
- exklusive Zugriffsanforderungen (X-Sperren)
- anfordernde TA besitzt bereits Sperren auf Datenobjekten
- keine vorzeitige Freigabe von Sperren auf Datenobjekten

Lösungsmöglichkeiten:
- Timeout: Transaktion nach festgelegter Wartezeit auf eine Sperre zurücksetzen
- Verhütung
- Vermeidung: potentielle Deadlocks im Vorraus erkennen
- Erkennung: Wartegraph mit Zyklensuche, dann Zurücksetzen einer oder mehrerer am Zyklus beteiligter TA

12 Recovery

12.1 Protokollierung und Wiederherstellung

Transaktionsparadigma fordert:
- Alles-oder-Nichts-Eigenschaft von Transaktionen
- Dauerhaftigkeit erfolgreicher Änderungen

Vorraussetzung:
- Sammeln redundanter Informationen während Betrieb (Logging)
- Mechanismen zur Wiederherstellung des jüngsten transaktionskonsistenten DB-Zustands (Recovery) offene Transaktionen rückgängig machen, abgeschlossene Transaktionen ggf. wiederholen

Globales Ziel: Erhaltung der physischen und logischen Konsistenz der Daten:
- physische Konsistenz: Korrektheit der Speicherungsstrukturen
- Logische Konsistenz: Korrektheit der Dateninhalte

12.2 Recovery-Klassen

- Partial Undo (R1): nach Transaktionsfehler, Zurücksetzen der veränderten Daten in den Zustand zu Beginn der (einen) Transaktion
- Partial Redo (R2): nach Systemfehler, Wdh. aller verlorengegangener Änderungen von abgeschl. TAs
- Global Undo (R3): nach Systemfehler, Zurücksetzen aller durch den Ausfall abgebrochenen Transaktionen
- Global Redo (R4): nach Gerätefehler, Einspielen einer Archivkopie

12.3 Einbringungsstrategien

Problem: Datenbankpuffer ⇒ Modifizierung der Pufferersetzung
Wann werden Daten aus dem Puffer auf die Platte geschrieben?
- Steal: bei Verdrängung aus dem Puffer, auch schon vor Ende einer Transaktion
- NoSteal: frühestens am Ende einer Transaktion
- NoForce: bei Verdrängung aus dem Puffer,
- Force: spätestens am Ende einer (erfolgreichen) Transaktion

Wie werden geänderte Daten aus dem Puffer auf die Platte geschrieben?
- NotAtomic: direkte Einbringstrategie (update in place), ist nicht ununterbrechbar zu machen
- Atomic: indirekte Einbringstrategie, ununterbrechbares Umschalten "von alt auf neu"

12.4 Protokolldaten

Was wird in die Protokolldateien geschrieben?

Welche Form?	Zustände	Übergänge
logisch	?	Änderungsoperationen (SQL)
physisch	Before-Images, After-Images	EXOR-Differenzen

Wann wird in die Protokolldatei geschrieben?
- Undo-Information: Write-ahead-Log, geschrieben bevor zugeh. Änderungen in Datenbestand eingebracht
- Redo-Information: WAL, geschrieben, bevor Abschluss d. TA an Programm/Benutzer gemeldet wird

Zustände vor bzw. nach einer Änderung werden protokolliert: Before-Image / After-Image

12.5 Sicherungspunkte

Maßnahmen zur Begrenzung des Redo-Aufwands nach Systemfehlern
Direkte (alle geänderten Seiten in DB) oder indirekte (Log-Datei) Sicherungspunkte:
- Transaction-Oriented Checkpoint: direkt nach Transaktionsende in DB (hohe Systembelastung)
- Transaction-Consistent Checkpoint: alle Änderungen erfolgreicher TAs, Lesesperre auf ganzer DB (Verzögerung)
- Action-Consistent Checkpoint: zum Zeitpunkt des Sicherungspunkts keine Änderungsoperationen aktiv

12.6 Recovery nach Systemfehler

bei direkter Seitenzuordnung: materialisierte DB unvorhersehbar → phys. Logging-Verfahren
bei indirekter Seitenzuordnung: materialisierte DB = Zustand des letzten erfolgreichen Einbringens

3-phasiger Ansatz:

- Analyse-Lauf: vom letzten Checkpoint bis zum Log-Ende
- Redo-Lauf: Änderungen der Gewinner-Transaktionen ggf. wiederholen
- Undo-Lauf: Rücksetzen der Verlierer-Transaktionen durch Rückwärtslesen des Logs bis zum BOT-Satz der ältesten Verlierer-Transaktion

12.7 Geräte-Recovery

Ziel: Wahrscheinlichkeit eines Gerätefehlers so weit wie möglich reduzieren
Datenbankrekonstruktion basierend auf:
Archiv-Log (Full/Incremental Backup) und temporärer Log

13 All together...