FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG

Implementierung von Datenbanksystemen

Vorlesungsmitschrift

Patrick Cerny

WS 2011/12

Semester 5

Inhaltsverzeichnis

1. Einführung **4**
1.1. Schichtenbildung 4
1.2. Datenbanktechnologie 4

2. Dateiverwaltung **5**
2.1. Physische Speichergeräte 5
2.2. Logische Speichergeräte 5
2.3. Dateiverwaltung eines Betriebssystems 5
2.4. Codeschnipsel 6
2.5. Referenzen 6

3. Sätze **7**
3.1. Anhang 7
3.2. Satzstrukturierte Daten 7
3.3. Sequenzielle Satzdatei 7
3.4. Direktzugriff auf Sätze 7
3.5. Codeschnipsel 8
3.6. Referenzen 8

4. Schlüsselzugriff - Teil 1 **9**
4.1. Satzzugriff über Schlüsselwerte 9
4.2. Streuspeicherung (Hashing) 9
4.3. Virtuelles Hashing 9
4.4. Lineares Hashing 9
4.5. Code-Schnipsel 10
4.6. Referenzen 10

5. Schlüsselzugriff - Teil 2 **11**
5.1. B-Bäume 11
5.1.1. B*-Baum 11
5.2. Bitmap-Index 11
5.3. Primär- und Sekundär-Organisation 12
5.4. Code-Schnipsel 12
5.5. Referenzen 12

6. Puffer **13**
6.1. Pufferverwaltung 13
6.2. Dienste einer Pufferverwaltung 13
6.3. Fehlersituationen 13
6.4. Seitenzuordnung 14
6.5. Schattenspeicher 14

7. Programmzugriff **16**
7.1. Programmzugriff 16
7.2. Unterprogrammaufruf 16

8. Transaktionen 18
8.1. Programmfehler . . . 18
8.2. Systemfehler . . . 18
8.3. Gerätefehler . . . 18
8.4. Die Transaktion . . . 19

9. Speicherung von Tupeln und Relationen 21
9.1. Speicherung von Tupeln in Sätzen . . . 21
9.2. C-Store . . . 21
9.3. Referenzen . . . 22
9.4. Notizen und Fragen . . . 22

10. Anfrageverarbeitung 23
10.1. Desktriptive Anfragespezifikation . . . 23
10.2. Aufteilung der Anfrageverarbeitung (Query Processing) . . . 23
10.3. Interndarstellung einer Anfrage . . . 23
10.4. Operatorbaum . . . 24
10.5. Standardisierung, Vereinfachung und Restrukturierung . . . 24

11. Relationale Operatoren 25
11.1. Anfrageoptimierung . . . 26
11.2. Erstellung und Auswahl von Ausführungsplänen . . . 26
11.3. DB-Katalog und statistische Kenngrößen . . . 26

12. Synchronisation 27
12.1. Transaktion . . . 27
12.2. Serialisierbarkeit . . . 27
12.3. Implementierungsmethoden . . . 27

13. Recovery 29
13.1. Einbringstrategien . . . 29
13.2. Protokolldaten . . . 30
13.3. Allgemeine Restart-Prozedur . . . 30

1. Einführung

Definiton 1 (Datenabstraktion)**.**
Speichern und Wiedergewinnen von Daten, ohne Kenntnis der Details der Speicherung.

Notwendigkeit generischer Software (Schema, Anfragen), um Schnittstellen zu ermöglichen.

1.1. Schichtenbildung

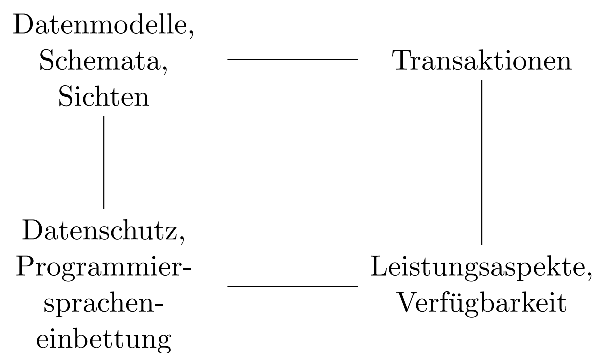
Definiton 2 (Schicht(en))**.**

Grundlegende Eigenschaften:

1. *Jede Hierarchieebene kann als abstrakte oder virtuelle Maschine aufgefasst werden (Wiederverwendbarkeit der Schichten)*
2. *Prinzip der Datenkapselung*
3. *Vererbung sämtlicher Funktionen*
4. *Systematische Abstraktion nach oben hin*

1.2. Datenbanktechnologie

DBS-Konzepte.



DBS setzen v.a. auf Datenunabhängigkeit, Datenintegration und Mehrbenutzerbetrieb.

Definiton 3 (DBS)**.**
Software, die die Daten entsprechend den vorgegebenen Beschreibungen abspeichert, auffindet oder weiter Operationen mit den Daten durchführt. DBS-Charakteristika:

- *vielseitig verwendbar*
- *gleichzeitig nutzbar*
- *ausfallsicher, ...*

2. Dateiverwaltung

Speichermedien der unteren Schichten sind physische Speichergeräte, höher liegende Schichten arbeiten auf logischen Speichergeräten. Magnetspeicher für Datenhaltung mit Abstand am wichtigsten! Dabei realisieren Betriebssysteme einen Verzeichnisdienst zur Verwaltung von Dateien mit umstrukturiertem Inhalt.

2.1. Physische Speichergeräte

- Wahlfreier Zugriff auf einen Slot-Inhalt (=Block): $\sim 10ms$
- kostengünstig, sequentieller Zugriff

2.2. Logische Speichergeräte

- Erhöhung der Störsicherheit
- Verwaltung und Aufruf durch eigene Instanz: Gerätetreiber
- Phys. Laufwerk in mehrere logische Platten aufteilen (Partition)

Starre Aufteilung der Platte führt zu ungenutztem Platz. Blöcke der Platte von allen Anwendungsprogrammen lesbar (kein Schutz), d.h. zahlreiche Fehlermöglichkeiten.

2.3. Dateiverwaltung eines Betriebssystems

Einrichten einer Indirektion: Dateiverwaltung

- Direktes Ansprechen der Blöcke über eine laufende Nummer
- Dynamisch erweiterbar um zusätzliche Blöcke

Dateikatalog als Verwaltungsdatenstruktur für Dateien, zusätzlich: Freispeicherverwaltung zur Lieferung unbenutzter Blöcke auf einer Platte. Anwendungsprogramme, die blockorientierten Dateizugriff verwenden, sind unabhängig von verwendeten Speichergeräten.

- Datei als abstrakte Sicht (virtuelle, logische Platte) auf diverse Abschnitte (Slots) eines oder mehrerer Plattengeräte
- Speichergerät kann reorganisiert oder ausgewechselt werden, ohne Nebenwirkung auf Programme
- Einstag über Dateinamen liefert physische Slot-Adresse

Katalogeintrag einer Datei per Extent-Tabelle. Repräsentiert ein zusammenhängendes (physisch sequentielles) Teilstück einer Datei, meist zur Dateierweiterung gedacht. Ein Extent besteht dabei aus mehreren Blöcken. Ein Eintrag der Extent-Tabelle ist definiert durch ein Array von [Slot-Adresse; Zahl der von dieser Adresse sequenziell belegten Slots].

2.4. Codeschnipsel

Listing 1: Lese-Operation

```
int Device::readBlock ( int CylinderNo, int TrackNo, int SlotNo,
    char *BlockBuffer )
```

Listing 2: Schreib-Operation

```
int Device::writeBlock ( int CylinderNo, int TrackNo, int SlotNo,
    char *BlockBuffer )
```

Listing 3: Zugriffsoperationen (Dienste)

```
BlockFile::BlockFile ( char *FileName, char Mode,
    int *BlockSize )
    ⋮
BlockFile::~BlockFile ()
```

Listing 4: Weitere Operationen

```
int BlockFile::append (int NumberOfBlocks )

int BlockFile::write ( int BlockNo, char *BlockBuffer )

int BlockFile::read ( int BlockNo, char *BlockBuffer )

int BlockFile::size ()

void BlockFile::drop ( int NumberOfBlocks )
```

2.5. Referenzen

[1]
http://www.speicherverwaltung.datenbank-wissen.de/tablespace.htm

3. Sätze

3.1. Anhang

Satzadresse über Indirektion: Verwaltung eines Feldes (ersten k Blöcke einer Datei), das zu jeder Satznummer (Feldindex) Blocknummer und Byte-Position enthält.

- Satzzugriff erfordert nun zwei Blockzugriffe
- Einfügen und Löschen per Speicherallokation und Valid-Bit
- Verschieben eines Satzes führt zur Änderung des Feldeintrags (Persistenz)
- Verschieben eines Satzes innerhalb eines Blocks wird lokal durch blockinternen Zuordnungstabelle geregelt
- Sog. Database Key Translation Table (DBTT) mit Database Key (DBK) zur Anwahl der Blocknummer

Aufbewahrungsmethodik der Sätze durch Verkettung (lineare (verkettete) Listen + Gruppenbildung)

3.2. Satzstrukturierte Daten

- Block: Einheit des Transports zwischen Platte und Hauptspeicher
- Entkopplung: Blöcke (phys. Satz) ↔ Datenstruktur der Anwendung (log. Satz)

Ein DBS besteht aus Datensätzen (Record bzw. Struktur), welche aus verschiedenen Feldern (Data Item) bestehen. Ein Feld hat einen Namen und einen Wert. Ein DBVS bearbeitet Datensätze (STORE, MODIFY, DELETE). Ein Block besteht i.d.R. aus mehreren Sätzen. Reihenfolge der Speicherung von Sätzen ist beliebig (Menge!).

3.3. Sequenzielle Satzdatei

- Schreibreihenfolge = Abspeicherungsreihenfolge = Lesereihenfolge
- Keine Unterstützung von (i) wahlfreiem Lesen, (ii) Einfügen und Ändern eines Satzes

Stapelverarbeitung als typische Bearbeitungsform für seq. Daten. Dadurch wird Blockunabhängigkeit realisiert. Zum Optimieren können Dateipuffer verwendet werden, die (mindestens) einen Block der Datei aufnehmen können, in die Sätze herausgeholt oder hineingepackt werden (Blockpuffer), d.h. Sätze werden gepuffert. n-Blockpuffer nennt man Pufferrahmen (Kacheln) → Umsetzung von Prefetching.

3.4. Direktzugriff auf Sätze

Gesucht: Flexiblerer Ansatz bei seq. Dateien (d.h. Einfügen, Löschen und Ändern einzelner Sätze).

Annahme: Direkter Zugriff auf Sätze per Satzadresse (eindeutig, unveränderlich), variable Satzlänge.

Realisierung durch TID-Konzept (Tuple IDentifier): Satzadressierung über Indirektion innerhalb der Blöcke. Hilfstruktur aus Anfangsadressen (Byte-Positionen) aller Sätze im Block. Satzadresse als Paar aus Blocknummer und Index des Felds. Freie Verschiebungsmöglichkeiten der Sätze innerhalb des Blocks. Bei Überlauf Verlagerung des Satzes in einen anderen Block (Verweis mit neuer Satzadresse des neuen Blocks, TID bleibt erhalten). Byte-Positionsindex wird mit jeder Verschiebeoperation angepasst.

System entscheidet über Abspeicherungsreihenfolge (Menge der Sätze)!

3.5. Codeschnipsel

Listing 5: Seq. Zugriffsoperationen

```
1 SeqRecordFile::SeqRecordFile ( char *FileName, char Mode, int *RecLength
      )
2
3 void SeqRecordFile::read (int *RecordLength, char *RecordBuffer )
4
5 void SeqRecordFile::write (char *RecordBuffer, int RecordLength )
6
7 SeqRecordFile::~SeqRecordFile ()
```

Listing 6: Direkte Satzdateien-Zugriffsoperationen

```
1 DirectRecordFile::DirectRecordFile ( ... )
2
3 RecordAddress DirectRecordFile::insert ( char *RecordBuffer, int
     RecordLength )
4
5 void DirectRecordFile::read ( RecordAdress Adress, char *RecordBuffer,
6    int *RecordLength )
7
8 void DirectRecordFile::modify ( RecordAdress Adress, char *RecordBuffer,
9    int RecordLength )
10
11 void DirectRecordFile::delete ( RecordAdress Adress )
12
13 void DirectRecordFile::first ( char *RecordBuffer, int *RecordLength )
14
15 void DirectRecordFile::next ( char *RecordBuffer, int *RecordLength )
```

3.6. Referenzen

[1] http://de.wikipedia.org/wiki/Netzwerkdatenbankmodell

[2] http://de.wikipedia.org/wiki/Datenbank

4. Schlüsselzugriff - Teil 1

4.1. Satzzugriff über Schlüsselwerte

- **Ziel**: über Inhalt des Satzes zugreifen zu können
- Realisiert durch (Such-)Schlüssel

Schlüsselzugriff erfordert neue Hilfsstrukturen, z.B. Sätze in effizienter Reihenfolge, d.h. gleichmäßig, abspeichern.

4.2. Streuspeicherung (Hashing)

Vorteil: Nur eine Berechnungsfunktion notwendig, keine Hilfsstrukturen! Berechnung der Hashfunktion mit dem Divisions-Rest-Verfahren, d.h. $h(k) = k \mod q + 1$ mit k als Schlüsselwert, q als # Buckets und $h(k)$ die errechnete Blocknummer.

Ziel: Berechnung der Speicherposition aus dem Schlüsselwert.

Umsetzung: Blöcke werden mit hashing "gefüllt" (Buckets). Dadurch wird der Suchaufwand innerhalb des Blocks vernachlässigbar. Aufgabe der Hashfunktion ist es, den Schlüsselwert in eine Blocknummer umzurechnen. Kollisionen sind bis zu einem Gewissen Grad erwünscht: mehrere Sätze passen in ein Bucket. **Aber**: Überlaufgefahr, da Berechnungsfunktion nicht optimal gewählt werden kann!

Lösung:

1. Open Adressing: Ausweichen auf Nachbar-Buckets in festgelegter Reihenfolge
2. Overflow-Buckets: Spezielle Überlauf-Buckets (gespeichert per Verkettung)
3. Nachteil beider Varianten: Zwei Blockzugriffe statt eines

Dynamische Speicherung zur Verwaltung der Hashstruktur notwendig, denn Speicherplatz muss im voraus belegt werden. Idee: online Reorganisation während der Einfügungen und Löschungen, d.h. sobald Belegungsfaktor $\beta >$ als Schwellenwert α wird die Menge der Buckets vergrößert. Dies erfolgt durch die Wahl einer neuen Hashfunktion und die Anschließende Umspeicherung des Datensatzes (inkl. Überläufer) auf die neuen Buckets.

4.3. Virtuelles Hashing

Wird der Schwellwert α überschritten, folgt eine schlagartige Vergrößerung des ursprünglich q_n großen Speichers auf q_{2n} bzw. $q_{4n}, \ldots$. Hier gibt es keine Überläufe.

4.4. Lineares Hashing

Anfänglich m_0 Buckets. Wenn α überschritten wird: neuen Bucket hinten anfügen und den Bucket, auf dem Positionszeiger p steht, aufteilen. Wir verwenden eine Folge von Hashfunktionen um die Position des Eintrags zu ermitteln (zuerst grob, d.h. diejenige Hashfunktion mit kleinem Modulo, dann fein). Überläufe sind (stark

begrenzt) möglich und werden in einer verketteten Liste gespeichert. Eine Inkrementation des Index i der Hashfunktion h_i, d.h. h_{i+1}, bedeutet jeweils eine Verdoppelung der Dateigröße. Gesplittet wird wie beim virtuellen Hashing (Belegungsfaktor $\beta = \frac{N}{(q \cdot 2^L + p) \cdot b}$).

4.5. Code-Schnipsel

Listing 7: Hashing

```
1 void HashRecordFile::insert ( char *RecordBuffer, int RecordLength, char
      *KeyValue )
2
3 char *HashRecordFile::readKey ( char *KeyValue, int RecordLength )
4
5 void HashRecordFile::modifyKey ( char *OldKeyValue, char *NewKeyValue )
```

Ist der Schlüssel nicht eindeutig, müssen gefundene Sätze mit `getNext` o.ä. einzeln abgerufen werden!

4.6. Referenzen

[1]
http://pages.cs.wisc.edu/~dbbook/openAccess/thirdEdition/slides/slides3ed-english/Ch11_Hash_Index.pdf

[2]
http://www.informatik.uni-jena.de/dbis/lehre/ws2010/dbsem/Vortraege/DynamischesHashing.pdf

[3]
http://en.wikipedia.org/wiki/Hash_table

[4]
http://compgeom.cs.uiuc.edu/~jeffe/teaching/373/notes/06-hashing.pdf

5. Schlüsselzugriff - Teil 2

5.1. B-Bäume

Ausgangspunkt: Binäre Such-Bäume.

Lösungsansatz unzureichend (zu viele Blockzugriffe) $\Longrightarrow$ Zusammenfassung bestimmter Sätze in einem Block (Mehrweg-Baum, d.h. jeder Knoten $\Leftrightarrow$ Block).

Ergebnis nennt man B(lock)-Baum. Ein B-Baum enthält folgende Einträge:

- K_i: Schlüsselwert
- D_i: Datensatz
- P_i: Zeiger auf den Nachfolgeknoten (Blocknummer). Dabei gilt: $P_0 \leq K1$ oder $P_i \geq K_i$.

Einfügen eines Satzen im (bereits vollen) B-Baum mit Hilfe des sog. Splitt: Der neue Blattknoten wird eingefügt, indem die 2k+1 Sätze gemäß Sortierordnung halbe-halbe aufgeteilt werden. Im Falle des Splittens des Wurzelknotens (und nur dann!), erhöht sich der B-Baum um 1. Dadurch gewährleisten wir einen balancierten Baum.
Beim Löschen im B-Baum muss auf die Balance des Baums geachtet werden, d.h. bei Bedarf erfolgt Mischen (Unterlauf) oder Splitten (Überlauf) *5[12]ff.*.

5.1.1. B*-Baum

Unterschied:

- Alle Sätze (bzw. Schlüsselwerte mit TIDs) werden in den Blattknoten abgelegt
- Innere Knoten enthalten nur noch Verzweigungsinformation, keine Daten
- $\Longrightarrow$ Sortierung der Sätze auf Blattebene

Was heißt das für das Einfügen/Löschen?

- Einfügen: Entweder normaler Splitt (wie beim B-Baum) oder neue Variante der Neuverteilung der Einträge unter Berücksichtigung eines oder mehrerer benachbarter Knoten
- Löschen: Standardmäßig durch Mischen, kann ebenfalls zum Unterlauf (d.h. # Einträge ¡ k?) führen (*siehe 5[26]*)

5.2. Bitmap-Index

Insgesamt: B-Bäume (sowie Hashing) sinnvoll für Suchschlüssel mit hoher Selektivität (Grenztrefferrate: 5

Idee: Jeder Schlüsselwert erhält in der Bitliste einen Wert $\in \{0, 1\}$, d.h. der Schlüssel hat im Satz den Wert, zu dem die Liste gehört.

Indexgröße: (#Sätze) $\times$ (#Ausprägungen) Bits

Durch die Selektivität entsteht einfache und effiziente logische Verknüpfbarkeit *5[32]*.

5.3. Primär- und Sekundär-Organisation

Primär-Organisation

- Bestimmt Speicherung der Sätze selbst (Blocknr.)
- Sequenzieller-, direkter Zugriff oder über Schlüssel

Sekundär-Organisation

- Verweist nur auf Sätze, die nach beliebigen anderen Kriterien abgespeichert wurden
- Nur möglich, wenn Primär-Organisation Direktzugriff auf einzelnen Satz unterstützt (Satzverweis). Im B- bzw. B*-Baum bei D_i an Stelle des Satzes nur Satzadresse eintragen.
- Beim Hashing in den Buckets nur (Schlüsselwert, Satzadress)-Paare

$\Longrightarrow$ Genau eine Primär-Organisation pro Datei, aber mehrere Sekundär-Organisationen möglich!

Realisierung: Eine Datei für die Sätze selbst, für jede Sekundär-Organisation ebenfalls eine eigene Datei. Wir benötigen einen (eindeutigen) Primärschlüssel, bei dem jeder Wert in höchstens einem Satz vorkommen darf (z.B. Kontonummer, Kundennummer, etc.). Der Index über Primärschlüssel muss allerdings nicht unbedingt die Primär-Organisation sein!

5.4. Code-Schnipsel

Listing 8: Zugriffsoperationen B-Baum, B*-Baum, Bitmap

```
void KeyedRecordFile::insert ( char *RecordBuffer, int RecordLength,
    char *KeyValue );

char *KeyedRecordFile::readKey ( char KeyValue, int *RecordLength );

void Keyedrecordfile::modifyKey ( char *OldKeyValue, char *NewKeyValue )
    ;
```

D.h.: Entscheidend ist der Zugriff über einen Schlüssel, nicht die Realisierung über Hashing, B-Baum, etc. ... $\Longrightarrow$ weitere Stufe von Datenunabhängigkeit (Datenstruktur-/Speicherungsstruktur-Unabhängigkeit)!

5.5. Referenzen

6. Puffer

6.1. Pufferverwaltung

Nun: Sequentieller und diekter Zugriff auf Sätze über Satzadresse oder Schlüssel.

Sogenannter (Buffer-) Frame zur Aufnahme eines Blocks vorgesehen (virtuell). Zugriff auf Block i ist ein logischer Zugriff:

- Block bereits im Puffer
- Block muss von Platte gelesen werden (physischer Zugriff, Einlagerung)

Problem: Einlagern verdrängt anderen Block aus dem Puffer $\Longrightarrow$ Ersetzungsstrategie notwendig:

- FIFO: Bewertet nur Alter
- LFU: Bewertet nur Häufigkeit
- LRU: Bewertet Alter seit dem letzten Zugriff (sehr gut, aber sehr aufwendig!)
- CLOCK: LRU Prinzip, allerdings mit Benutzt-Bit, der angibt, nach der wievielten Suche eine Seite ersetzt werden soll.

6.2. Dienste einer Pufferverwaltung

Einkapselung der Pufferverwaltung:

Listing 9: Einkapselung der Pufferverwaltung

```
char *Buffer::fix ( BlockFile File, int BlockNo, char Mode );

void Buffer::unfix ( char *BufferAdress );
```

`Mode` gibt an, ob Block nur gelesen oder auch geändert werden soll

`fix` schützt Block vor Verdrängung, bis Bearbeitung abgeschlossen ist (auch ein leerer Block muss mit `fix` im Puffer bereitgestellt werden!)

`unfix` gibt Block im Puffer zur Ersetzung frei

6.3. Fehlersituationen

Betriebssystem-Absturz, Stromausfall, etc. führt zu Datenverlust! Blöcke auf der Platte dadurch inkonsistent (alte und neue Blöcke passen nicht zusammen!).
Beispiel: B-Baum nach Splitt

- Neuer Knoten bereits auf Platte geschrieben aber übergeordneter Knoten noch alt
- $\Longrightarrow$ Neuer Knoten wird nicht beachtet!

Vorbeugung durch sogenannte Einbringstrategien:

- alte Blockinhalte auf Platte beibehalten, neue Blockinhalte in andere Slots schreiben
- am Schluss: auf einen Schlag (ununterbrechbar) umschalten von alt auf neu
- bei Fehler: neue Blöcke ignorieren, alten Zustand wiederherstellen
- **Einbringen:** Ablegen auf einem nicht-flüchtigen Speicher
- Trennung in **Block** und **Seite** (= Block im Puffer)
- Anwender arbeiten nur noch mit Seiten (flexible Zuordnung zu Blöcken)
- **Segment:** logischer Adressraum, entspricht der Datei. Ein Segment ist eine Folge von Seiten. Zuordnung kann sein 1:1, N:1, 1:N (# Seiten der Segmente:Blöcke einer Datei)

6.4. Seitenzuordnung

- direkte Seitenzuordnung: aufeinanderfolgende Seiten werden auf aufeinanderfolgende Blöcke einer Datei abgebildet
  - keine Fragmentierungsprobleme
  - funktioniert nur für N:1 oder 1:1
  - **update-in-place:** bei Verdrängung ersetzt eine Seite genau den Block, aus dem sie beim Einlagern in den Puffer gelesen wurde $\Longrightarrow$ keine Wiederherstellung nach Ausfällen möglich
- indirekte Seitenzuordnung: mehr Flexibilität, benötigt allerdings Hilfstruktur (Array) mit Blocknummer zu jeder Seite
  - für jedes Segment existiert eine Seitentabelle, die für jede Seite einen Eintrag mit der aktuellen Blockzuordnung besitzt
  - Für eine Datei D existiert zusätzlich eine Bitliste Map, die ihre aktuelle Belegung beschreibt, d.h. für jeden Block angibt, ob er momentan eine Seite enthält oder nicht
  - bei Verdrängung wird eine Seite in einen freien Block geschrieben. Der urpsrüngliche Block bleibt unverändert $\Longrightarrow$ nach Hauptspeicherverlust ist eine konsistente Datenbank in den alten Blöcken verfügbar

Wann und wie wird man die alten Blöcke los? $\Longrightarrow$ man muss wissen, wann alle zusammenhängenden Blöcke gespeichert wurden.

6.5. Schattenspeicher

- Inhalte aller Seiten eines Segments werden in einem Sicherungsintervall Δt in einem konsistenten Zustand unverändert gehalten
- basiert auf **Sicherungspunkten:** Ein Sicherungspunkt besteht aus belegten Seiten, Seitentabelle, Bitliste
- im Fehlerfall: segmentorientiertes Zurückgehen auf letzten Sicherungspunkt
- Funktionsweise siehe 6[24]ff.

- Cluster-Eigenschaften von Blöcken gehen verloren, d.h. Konzept ist nur für kleinere DB geeignet, für größere DB ist direktes Einbringen besser (also praktisch für alle ...)

7. Programmzugriff

Benutzer arbeitet mit "virtuellenÖbjekten Segment (Datei) und Satz:

- Segment = Menge von Sätzen
- Satz = variable lange Folge von Bytes, evtl. plus Schlüssel

Operationen:

- sequenzielles Lesen
- Direktzugriff über Satzadresse
- Direktzugriff über Schlüssel

Dadurch: Unabhängigkeit der Programme von Organisationsformen (Segmenttypen) und Indexen! Noch mächtigere Operationen durch Boolesche Ausdrücke, etc.

Weitere Abstraktion:

- von Dateien zu Relationen (Klassen)
- von Sätzen zu Feldern (Objekte)
- von Feldern zu Attributen

7.1. Programmzugriff

- Precompiler transformiert SQL-Anweisungen in normales C (`exec sql <SQL-Anweisung> ;`)
- Programmvariablen werden zur Unterscheidung von Attributen durch vorangestellten Doppelpunkt ":" gekennzeichnet
- da `select`-Anweisung Mengen liefert $\Longrightarrow$ Tupelzeiger notwendig (`declare c1 cursor for`)
- Öffnen des Zeigers mit `open c1`, holen von Tupel mit `fetch c1 into`, Schließen mit `close c1`
- erlaubt Erzeugung von Zugriffsmodulen, dadurch SQL-Anweisungen schon zur Übersetzungzeit durch Precompiler an DBS übergeben; es kann analysiert, optimiert, etc. werden

7.2. Unterprogrammaufruf

- Call-Level-Interface (CBI) eines DBS (`call DBS (‘‘select ... from ... where’’);`)
- z.B. mit JDBC (J=Java) zur Verarbeitung von Anfrageergebnissen:
  - Java kann Mengen nicht direkt übersetzen $\Longrightarrow$ Iteration über eine Menge mit Cursor
  - Schnittstelle "Statement": `executeQuery, executeUpdate, execute`
  - optimierte Anfragen mit `PreparedStatement`: schon beim Erzeugen SQL-Anweisung angeben $\Longrightarrow$ wird sofort analysiert, übersetzt optimiert, in Zwischencode überführt, ..., d.h. man muss dann nur noch die Parameterwerte setzen (`setBoolean, setString`)

- **gespeicherte Prozeduren:** Schnittstelle `CallableStatement`
  * Idee: Anfrage in der Datenbank ablegen, mit einem Namen und Parametern versehen und dort aufrufen ("Stored Procedures")
  * $\Longrightarrow$ Analyse und Optmierung nur noch einmalig, d.h. explizite Optmierung

8. Transaktionen

8.1. Programmfehler

Datenbank-Anwendungsprogramm stürzt ab

- Division durch Null, unzulässige Adressierung, ...
- Abbruch von außen (z.B. cancel, kil oder Überschreiten einer CPU-Zeit-Vorgabe)

$\Longrightarrow$ Hauptspeicherinhalte dieses Programms verloren $\leftarrow$ unbekannter Zustand.

Puffer leeren, Platte muss per Hand bereinigt werden
$\Longrightarrow$ **Aufwand! Zeit! Daten blockiert für andere!**

8.2. Systemfehler

DBVS fällt aus oder BS fällt aus oder Hardware-Fehler etc.

- Programm war evtl. schon fertig, d.h. Änderungen vollständig ausgeführt
- leider bisher nur im Puffer

Ergebnis: Pufferinhalte sind verlorengegangen, Daten auf Platte in unbekanntem Zustand

Alle Programme, die gerade liefen oder kurz zuvor fertig geworden sind, müssen von Hand nachvollzogen werden!
$\Longrightarrow$ **extrem** großer Aufwand!

8.3. Gerätefehler

z.B. Platte defekt ...

$\Longrightarrow$ Änderungen auf Platte verloren, d.h. DBVS und BS können icht mehr weiter machen

Lösung durch Formatieren oder BackUp aufspielen $\Longrightarrow$ **Auwand, Zeit, Daten blockiert!**

physische Konsistenz

- Korrektheit der Speicherungsstrukturen
- TIDs und Verweise müssen stimmen, sowie Vollständigkeit der Indexe

logische Konsistenz

- Korrektheit der Dateninhalte
- Alle Bedingungen des Datenmodells (Primärschlüsseleigenschaft, Referenzielle Integrität, benutzerdef. Assertions) sind erfüllt

Vollständig ausgeführte Datenbank-Operationen hinterlassen einen physisch konsistenten Zustand, vollständig ausgeführte Anwendungsprogramme einen logisch konsistenten Zustand.

Nach einem **Fehler:** Daten i.A. weder physisch noch logisch konsistent.

Ziel:

- Systemunterstützung zur Wiederherstellung des (physischen/logischen) Zustands der Daten nach einem Fehlerfall (**Backward/Forward Recovery**)
- benötigt sog. Sicherungs- und Protokollierungsmaßnahmen (**Logging**) auf Kosten von Speicher, E/A, CPU

8.4. Die Transaktion

- Folge von DB-Operationen, die, von einem logisch konsistenten Zustand ausgehend, die Datenbank wieder in einen logisch konsistenten Zustand überführt
- muss vom Anwender definiert werden
- DB-Aufruf commit: Anwendungsprogramm teilt dem DBVS mit, wann ein logisch konsistenter Zustand hergestellt ist (vgl. Banküberweisung)
- DB-Aufruf abort: Rückkehr in den Anfangszustand auf Wunsch des Anwenders
- $\Longrightarrow$ Datenbanktransaktion stellt eine logische Arbeitseinheit dar
- Zusammenfassung ovn aufeinanderfolgenden DB-Operationen
- überführen eine Datenbank von einem konsistenten Zustand in einen neuen konsistenten Zustand (physische und logische Konsistenz)
- innerhalb einer Transaktion kiönnen vorübergehend logisch inkonsistente Zustände auftreten
- Transaktionen sind atomar und ununterbrechbar (**Atomarität**)
- **Konsistenzerhaltung:** Voraussetzen eines konsistenten Zustand des Systems und hinterlässt nach erfolgreichem Abschluss wieder einen konsistenten Zustand
- **Vollständigkeit:** Alle erforderlichen Änderungen enthalten (so lange wie nötig)
- Mehrbenutzerbetrieb: Inkonsistenzen vermeiden (**Synchronisation**), d.h. läuft eine Transaktion, darf keine andere Transaktion von ihr bereits durchgeführte Änderungen lesen und benutzen (d.h. strikt serielle Ausführung, fiktiver Einbenutzerbetrieb, isolation); realisiert durch Blocking von Daten für andere (Transaktionen so kurz wie möglich und so lang wie nötig)
- **Dauerhaftigkeit:** Erfolgreich abgesdchlossen gemeldete Transaktionen sind dauerhaft (persistent)
- `exec sql commit` und `exec sql rollback` beenden die Transaktion

Verschiedene Fälle:

- Programm führt *nur eine* Transaktion aus: Scheitern $\Longrightarrow$ Programm ggf. mit gleichen Eingabedaten nochmal ausführen

- Programm führt *mehrere* Transkationen hintereinander aus: Erneute Ausführung dahin modifiziert, dass bereits erfolgreich abgeschlossene Transaktionen nicht wiederholt werden! (vgl. Banküberweisung)
- Beispiel siehe 8[25]ff.

9. Speicherung von Tupeln und Relationen

Von oben nach unten durch das Schichtenmodell: *Was passiert mit einer Anfrage?*

9.1. Speicherung von Tupeln in Sätzen

Sätze sind aus Feldern zusammengesetzt und werden durch den Systemkatalog verwaltet. Jedem Satz wird beim Abspeichern ein **Satztyp** zugeordnet (n:1 oder n:m). Satztypen sind Mengen von Sätzen mit gleicher Struktur (z.B. Tupel derselben Relation).

Annahme: Reihenfolge der Felder spielt keine Rolle!

Anforderungen:

- Speicherplatzeffizienz
- direkter Zugriff auf Felder
- Flexibilität: Hinzufügen von Feldern bei allen Sätzen, Löschen eines Feldes aus allen Sätzen
- Speicherstrukturen: Eingebettete Längenfelder mit Zeigern
  - fester Strukturteil: Felder fester Länge und Zeiger
  - variabel lange Felder ans Ende des Strukturteils legen
  - satzinterne Adresse aus Katalogdaten berechenbar
- Spaltenweise Speicherung:
  - gut für analytische Auswertungen (große Datenmengen), d.h. auf das Lesen hin optimiert
  - nur die Attribute einlesen, die gebraucht werden (kompakte Speicherung der Attributwerte)

9.2. C-Store

- Performanz durch spaltenweise Speicherung (Column-Store)
- Aufbau in (optimierten) Write-Storage und Read-Storage, wobei mit Hilfe eines **Tuple Mover** Daten vom WS in RS bewegt werden
- speichert Sammlung von Spalten (jeweils sortiert nach Attribut)
- Gruppe von Spalten = Projektionen
  - ein Attribut oder mehrere Attribute einer Tabelle (Duplikate bleiben erhalten)
  - z.B. `DEPT (dname,floor)`
- Rekonstruktion vollständiger Tupel muss möglich sein (mit Hilfe von Speicherschlüssel, Verbund-Indexe)
- Identifizierung von Tupel über sog. Speicherschlüssel (SK)
- **Verbund-Indexe:** M Segmente in T1 zu N Segmenten in T2 ergibt eine Sammlung von M Tabellen, je eine pro Segment von T1, mit Tupeln: `(s: SID in T2, k: SK in Segment s)`

- **verteilte Speicherung** von Segmenten und ihren Verbund-Indexen, dessen Allokation bei Bedarf K-sicher ist, wenn der Ausfall von K Knoten immer noch eine Überdeckung und Rekonstruktion von T erlaubt

Komprimierung abhängig von:

- Ordnung
- Anteil der Duplikate

Hierzu gibt es verschiedene sortierende Speichermethoden (Verbund-Indexe müssen dabei (leider) unkomprimiert gespeichert werden).

9.3. Referenzen

[1] http://de.wikipedia.org/wiki/Spaltenorientierte_Datenbank

[2] http://de.wikipedia.org/wiki/Projektion_(Informatik)#Projektion

[3] http://db.csail.mit.edu/projects/cstore/vldb.pdf

9.4. Notizen und Fragen

- Was sind Segmente?

10. Anfrageverarbeitung

Ziel: Mengenorientierter Zugriff (d.h. Reihenfolge ist egal) durch Abbildung von mengenorienter Operatoren auf Satzorientierte, sowie die Benutzung von Indexstrukturen.

10.1. Desktriptive Anfragespezifikation

- Anforderung: syntaktische Korrektheit, Integrität, Anfrageoptimierung
- **zentrale Aufgabe:** Umsetzung deskriptiver Anfragen in eine optimale Folge interner DBS-Operationen (an die Satzschnittstelle)
- optimierte Abfragen sind drastisch schneller (siehe Indexausnutzung, 10[10])

10.2. Aufteilung der Anfrageverarbeitung (Query Processing)

Anfrageverarbeitung (AV) ↔ Anfrageausführung (AA)

AV liefert Anfrageausführungsplan, arbeitet auf logischem DB-Prozessor (im Gegensatz zur AA, welcher auf physischem DB-Prozessor arbeitet). Zur AA wird der Anfrageplan dann tatsächlich zur Laufzeit ausgeführt. *(Übersicht: "Phasen der Anfrageverarbeitung" siehe 10[11])*

Phasen der Anfrageverarbeitung

- **lexikalische und syntaktische Analyse:** Überprüfung auf korrekte Syntax (Parsing), Erstellen eines Anfragebaums
- **semantische Analyse:** Prüfung auf Gültigkeit der referenzierten Relationen und Attribute, Namensauflösung, Binden, Konversion in interne Darstellung
- **Zugriffs- und Integritätskontrolle:** Format- und Konvertierungskontrolle von Datentypen
- **Standardisierung und Vereinfachung:** Überführung des Anfragebaums in eine Normalform, Beseitigen von Redundanzen
- **Restrukturierung und Transformation (=Verbesserung):**
  - algebraische Optimierung (Rekonstrukturierung) zur globalen Verbesserung des Anfragebaums
  - nicht-algebraische Optimierung (Transformation) durch Ersetzen ggf. Zusammenfassen der logischen Operatoren durch Planoperatoren
  - Auswählen der günstigsten Planalternative
- **Code-Generierung:** Generierung eines zugeschnittenen Programms für die vorgegebene (SQL-) Anfrage und Erzeugung eines ausführbaren Zugriffsmoduls (verwaltet in einer DBVS-Bibliothek)

10.3. Interndarstellung einer Anfrage

Relationale Algebra definiert relationale logische Operatoren:

- **Selektion:** Auswahl von Zeilen

- **Projektion:** Auswahl von Spalten
- **Kreuzprodukt:** Konkatenation jedes Tupels der einen Relation mit jedem ader anderen
- **Verbund:** Konkatenation derjenigen Tupel aus zwei Relationen, die eine Bedingung erfüllen
- **Mengenoperationen**
- ausführliche Beispiele siehe 10[16]ff.
- Mengenoperatoren auf zwei Relationen:
  - Kreuzprodukt: CROSS (R,S)
  - Verbund: JOIN (R,S,pred)
  - $R \cup S$ bzw. UNION (R,S)
  - $R \cap S$ bzw. INTERSECT (R,S)
  - $R \setminus S$ bzw. EXCEPT (R,S)

10.4. Operatorbaum

- effiziente Datenstruktur mit geeigneten Zugriffsfunktionen (prozedurale Darstellung einer deskriptiven, mengenorienten Anfrage)
- Knoten sind Operatoren der relationalen Algebra, Blattknoten sind Relationen
- gerichtete Kanten repräsentieren den Datenfluss

10.5. Standardisierung, Vereinfachung und Restrukturierung

- **Standardisierung:** Wahl einer Normalform (KNF, DNF, Pränex-Normalform (z.B. Auflösung geschachtelter `select`-Anweisungen)
- **Vereinfachung:** Redundanzen auflösen (Idempotenzregel, Ausdrücke mit leeren Relationen, Eliminierung gemeinsamer Teilausdrücke)
- **Restrukturierung:** äquivalente Umformung des Operatorbaums (Aussuchen der effizienteren Variante, Regeln siehe 10[22]f.)
  - Ziel: Zwischenergebnisse möglichst klein halten (d.h. v.a. Kreuzprodukt vermeiden)
  - vereinfachte Vorgehensweise (**Heuristik**): komplexe Verbundoperationen zerglegen in binäre Verbunde
  - Selektionen so früh wie möglich ausführen, d.h. Selektionen hinunterschieben zu den Blättern des Anfragebaums (dabei teure Duplikateliminierung vermeiden!)
  - Selektionen und Kreuzprodukt zu Verbund zusammenfassen, wenn das Selektionsprädikat Attribute aus den beiden Relationen verwendet
  - etc. ...

11. Relationale Operatoren

- Ersetzen logischer Operatoren durch **Planoperatoren**
- Minimierung der Größe der Zwischenergebnisse
- erkennen gemeinsamer Teilbäume
- ausführbar als parametrisierte Unterprogramm
- Ausgabe: ganze Relation, nächstes Tupel bzw. nächste n Tupel
- SQL erlaubt komplexe Anfragen über k Relationen (die in 1-2 Relationen aufgebrochen werden)

Selektion

- mehrere Planoperatoren
- Nutzung des Scan-Operator

Projektion

- typischerweise in Kombination mit Sortierung, Selektion oder Verbund durchgeführt

Sortierung

- erforderlich bei `order by` (Gruppierung, Duplikateliminierung)
- allerdings blockierend
- extern, d.h. Teilergebnisse müssen auf Externspeicher ausgelagert werden

Verbund

- teuer und häufig $\implies$ Optimierungskandidat!!!
- mögliche Zugriffspfade z.B. über Verbundattribute oder Selektionsattribute
- **Nested-Loop-Verbund**: Scan über S, für jeden Satz S scan über R ($\mathcal{O}(N^2)$) (*siehe 11[10]*)
- **Sort-Merge-Verbund**: Sortierung von R und S (Eliminierung nicht benötigter Tupel), anschließend schritthaltende Scans über sortierte R- und S-Relationen (Ausnutzen von Indexstrukturen) ($\mathcal{O}(N \log N)$

Hash-Verbund

Hauptspeicher für Zwischenergebnisse ausnutzen (häufiger Fall: **Gleichverbund**).

Idee: Tupel der einen Relation im HS ablegen, so dass sie über Verbundattribut schnell gefunden werden können. Organisation über **Hashing**.

Ablauf:

- äußere Schleife: abschnittsweises Lesen der kleineren Relation R und Aufteilen in p Abschnitte derart, dass jeder Abschnitt in den HS passt

- innere Schleife: für jeden Abschnitt R ← Überprüfung für jeden Satz von S mit P(S.SA), im Erfolgsfall Verbundoperation (*siehe 11[17]*), d.h. Verbundpartner S wird p-mal gelesen
- Optimierung durch zusätzliche Partitionierung von S ← d.h. Hashing von R erfolgt nun wertemäßig, nicht nach der Reihenfolge der Tupel

11.1. Anfrageoptimierung

Ziel: Von der Anfrage (WAS?) zur Auswertung (WIE?), d.h. Ermittlung des kostengünstigsten Auswertungswegs.
Problem: globale Optimierung i.A. aufwändig $\Longrightarrow$ Einsatz von Heuristiken.
Optimierungsziel: Durchsatzmaximierung, d.h. maximaler Output bei gegebenen Ressourcen, oder Minimierung der Ressourcennutzung für gegebenen Output. (Kosten sind z.B. Berechnungskosten, E/A-Kosten, Speicherungskosten etc.)

11.2. Erstellung und Auswahl von Ausführungsplänen

Eingabe: optimierter Anfragebaum, Ausgabe: optimaler Ausführungsplan.

Problem: sehr große Suchbäume bei komplexen Anfragen.
Vorgehensweise:

- Generieren aller vernünftigen, logischen Ausführungspläne und Vervollständigung dieser
- Bewertung der generierten Alternativen und Auswahl des billigsten Ausführungsplans
- Nachteil: bei einer Anfrage mit 15 Verbündung z.B. 10^{70} mögliche Ausführungspläne
- deshalb Ziel: Auffinden eines guten Plans (mittels Kostenberechnung) mit einer möglichst kleinen Anzahl generierter Pläne (*siehe 11[26]*)
- dazu: Statistiken sammeln und im DB-Katalog organisieren

11.3. DB-Katalog und statistische Kenngrößen

- Problem: Aktualisierung sehr aufwändig $\Longrightarrow$ kann zum Engpass werden
- periodische Neubestimmung der Statistiken (per Heuristiken), d.h.:
- Ermittlung (Abschätzen) eines Selektivitätsfaktors SF ($0 \leq SF \leq 1$), der die Anzahl an erwarteter Tupeln angibt, die das Prädikat p erfüllen item `SF(p) = card(SEL(R,P)) / card(R)` mit `card(X)` als Kardinalität

Fazit: Vorhandene Indexstrukturen können die Laufzeit dramatisch reduzieren, lohnt sich allerdings nur bei hoher Selektivität, d.h. bei sehr geringen Trefferraten. Ansonsten verwendet man einen Relationen-Scan.

12. Synchronisation

12.1. Transaktion

Zusammenfassung von aueinander folgenden DB-Operationen, die eine Datenbank von einem konsistenten Zustand wieder in einen konsistenten Zustand überführt (`commit`: Änderungen sind permanent, `abort/rollback`: bereits durchgeführte Änderungen werden zurückgenommen).

Eigenschaften:

- **A**tomicity, **C**onsistency, **I**solation, **D**urability (**ACID**)
- Atomicity: Alles-oder-nichts-Prinzip
- Consistency: Integritätsbedingungen werden eingehalten
- Isolation: Transaktionen laufen voneinander isoliert ab (d.h. ohne Zwischenergebnisse)
- Durability: Alle Ergebnisse erfolgreicher Transaktionen müssen persistent gemacht worden sein

Ziel: Erhaltung der Transaktionskonsistenz (= operationelle Integrität) im Mehrbenutzerbetrieb (**Synchronisation**).

D.h.: **Serialisierung** für Mehrbenutzerbetrieb notwendig (mit Scheduling!) ← "virtuelle" serielle Ausführung (logischer Einbenutzerbetrieb).

Mögliche Anomalien ohne Synchronisation:

- "lost updates" (*siehe 12[8]*)
- Abhängigkeit von nicht freigegebenen Änderungen (dirty read, dirty overwrite)
- inkonsistente Analysen

12.2. Serialisierbarkeit

Wenn alle Transaktionen seriell ausgeführt werden, dann bleibt die Konsistenz der Datenbank erhalten (**verzahnte Abläufe**).

Problem: nicht alle verzahnten Abläufe sind serialisierbar.

Abhängigkeitsgraph

- **Knoten:** einzelne Transaktionen
- **Kanten:** Abhängigkeiten zwischen zwei Transaktionen
- Serialisierbarkeit liegt vor, wenn der Graph keine Zyklen enthält

12.3. Implementierungsmethoden

Logischer Einbenutzerbetrieb durch sog. Sperren (**locks**) für Zugriffe auf Datenobjekte:

- **X** (eXclusive)-Sperre (= Schreibsperre)
- S (Shared)-Sperre (= Lesesperre)
- bei Zugriff auf Datenobjekt wird dieses vorher gesperrt, nach Ende der Transaktion wird Sperre wieder freigegeben (verwaltet in sog. **Kompatibilitätsmatrix**)
- **Sperrgranulat Tupel**:
  - viele oder alle Tupel einer Relation auf einmal sperren
  - große Sperrtabellen $\implies$ hohe Verwaltungskosten
- **Phantom-Problem:** Sperren nur auf existierende Tupel $\rightarrow$ da neue Tupel allerdings jederzeit eingefügt werden können, sind diese dann nicht gesperrt (= Phantome)
- **Lösung:** hierarchische Schachtelung der Datenobjekte (Baumstruktur)

Bei Sperren werden automatisch Unterknoten mitgesperrt $\implies$ Erweiterungen:

- **IS-Sperre**: falls auf untergeordnete Objekte nur lesend zugegriffen wird
- **IX-Sperre**: falls auf untergeordnete Objekte schreibend zugegriffen wird
- **SIX = S + IX**: sperrt das Objekt in S-Modus und verlangt auf tieferen Hierarchieebenen nur noch IX- oder X-Sperren für zu ändernde Objekte
- Baumstruktur $\rightarrow$ Top-Down beim Erwerb von Sperren, Bottom-Up bei der Freigabe von Sperren

Probleme bei der Sperrverwaltung:

- Permormanz wichtig $\rightarrow$ explizite, tupelweise Sperren führen zu umfangreichen Sperrtabellen = großer Zusatzaufwand
- starke Serialisierung $\implies$ lange Wartezeiten
- Engpässe $\rightarrow$ Deadlock-Gefahr bei zyklischen Wartebeziehungen

Mögliche Lösungen:

- **Timeout**
- Verhütung (**Prevention**): keine Laufzeitunterstützung zur Deadlock-Behandlung erforderlich
- Vermeidung (**Avoidance**): potentielle Deadlocks erkennen $\rightarrow$ Laufzeitunterstützung nötig
- Erkennung (**Detection**): explizites Führen eines Wartegraphen mit Zyklensuche

13. Recovery

- Alles-oder-Nichts-Eigenschaft von Transaktionen
- Dauerhaftigkeit erfolgreicher Änerungen

Voraussetzungen dafür:

- Logging, Protokollierung
- Wiederherstellungsmöglichkeit des jüngsten transaktionskonsistenten DB-Zustands (Recovery)
- **Ziel:** Erhaltung der physichen Konsistenz (= Korrektheit der Speicherungsstrukturen) und logischen Konsistenz (= Korrektheit der Dateninhalte) der Daten
- **Merke:** logische Konsistenz setzt physische Konsistenz voraus!

Recovery-Klassen

- Partial Undo: nach Transaktionsfehler (R1-Recovery)
- Partial Redo: nach Systemfehler (Verlust des Hauptspeichers, verlorerengegangene Änderungen nur Daten des Puffers) (R2-Recovery)
- Global Undo: nach Systemfehler (Verlust des Hauptspeichers, Zurücksetzen aller unterbrochenen Transaktionen) (R3-Recovery)
- Global Redo: nach Gerätefehler (R4-Recovery)

13.1. Einbringstrategien

Allgemein: Gültigkmachen von Datenobjekten in der Datenbank, so dass sie auch nach Fehlern benutzt werden können ("materialisieren").

Problem mit **Datenbank-Puffer**: direktes und indirektes Einbringen → Systemfehler führt zum Datenverlust!

WANN werden geänderte Daten aus dem Puffer auf die Platte geschrieben?

- **Steal:** bei Verdrängung aus dem Puffer schon vor dem Ende der Transaktion
- **NoSteal:** frühestens am Ende der erfolgreichen Transaktion (kein Undo erforderlich)
- **Force:** spätestens am Ende der erfolgreichen Transaktion (kein Partial Redo erforderlich)
- **NoForce:** erst bei Verdrängung aus dem Puffer, also nach dem Ende der Transaktion

WIE werden geänderte Daten aus dem Puffer auf die Platte geschrieben?

- **Atomic:** indirekte Einbringstrategie
- **NotAtomic:** direkte Einbringstrategie (update in place)

13.2. Protokolldaten

WAS wird auf die Protokolldateien geschrieben? → protokolliert werden nur die Informationen, die nach einem Systemausfall den jüngst möglichen konsistenten Zustand wiederherstellen können.

WANN wird auf die Protokolldatei geschrieben?

- UNDO-Information: muss geschrieben sein, bevor die zugehörigen Änderungen in den DAtenbestand eingebracht werden (WAL-Prinzip, “write ahead log”)
- REDO-Information: muss geschrieben sein, bevor der Abschluss der Transaktion an Programm bzw. Benutzer gemeldet wird

Physische Protokollierung

Protokolliert wird auf Seiten und Sätzen, und zwar folgende Zustände:

- alter Zustand: Before-Image (BI) für UNDO
- neuer Zustand: After-Image (AI) für REDO
- sehr Aufwändig $\Longrightarrow$ Ziel: Optimierung durch z.B. Sammlung und Pufferung im Hauptspeicher
- Reduzierung des REDO-Aufwands durch “checkpoints”:
  - **direkte Sicherungspunkte:** Ausschreiben und Einbringen aller geänderten Seiten in die Datenbank
  - **indirekte Sicherungspunkte:** Protokollierung von Statusinformationen in der Log-Datei
- weitere Verfahren:
  - TOC (Transaktionsorientierte Sicherungspunkte): geänderte Seiten einer Transaktion werden sofort nach Transaktionsende in die Datenbank eingebracht (kein REDO mehr notwendig, dafür hohe Belastung)
  - TCC (Transaktionskonsistente Sicherungspunkte): Einbringen aller Änderungen erfolgreicher Transaktionen unter Lesesperre auf ganzer DB (REDO und UNDO möglich)
  - ACC (Aktionskonsistente Sicherungspunkte): zum Zeitpunkt des Sicherungspunkts dürfen keine Änderungsoperationen aktiv sein (nur REDO möglich)
  - Crash-Recovery: Herstellung des jüngsten transaktionskonsistenten DB-Zustand aus materialisierter DB und temporärer Log-Datei

13.3. Allgemeine Restart-Prozedur

3-phasiger Ansatz

- **Analyse-Lauf:** vom letzten Checkpoint bis zum Log-Ende; Ermittlung der Gewinner- und Verlierer-Transaktionen
- **Undo-Lauf:** Rückwärtslesen des Logs; Rücksetzen der Verlierer-Transaktionen
- **Redo-Lauf:** Vorwärtslesen des Logs; Änderungen der Gewinner-Transaktionen ggf. wiederholen