IDB Skript

23. Juni 2014 - 10:24 Uhr

Dieses Skript zur Vorlesung Implementierung von Datenbanksystemen im Wintersemester 2013/14 wurde von untenstehenden Studenten erarbeitet. Es ist somit offensichtlich inoffiziell und erhebt weder einen Anspruch auf Korrektheit noch auf Vollständigkeit.

Christian Bay christian.bay@studium.fau.de

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einführung

1.1 Basiswissen

1. Warum ein Datenbanksystem benutzen?
   - vielseitig verwendbar
   - Mehrbenutzerbetrieb
   - redundanzfrei
   - ausfallsicher
   - leistungsfähig
2. Wozu dient Schichtenbildung in der Softwarearchitektur?
   - Höhere Ebenen werden einfacher, weil sie tiefere benutzen können
   - Änderungen in höheren Ebenen haben keinen Einfluss auf tiefere
   - tiefere Ebenen können getestet werden, bevor höhere lauffähig sind
   - Optimierungen von unteren Ebenen

   Schichtenmodell:

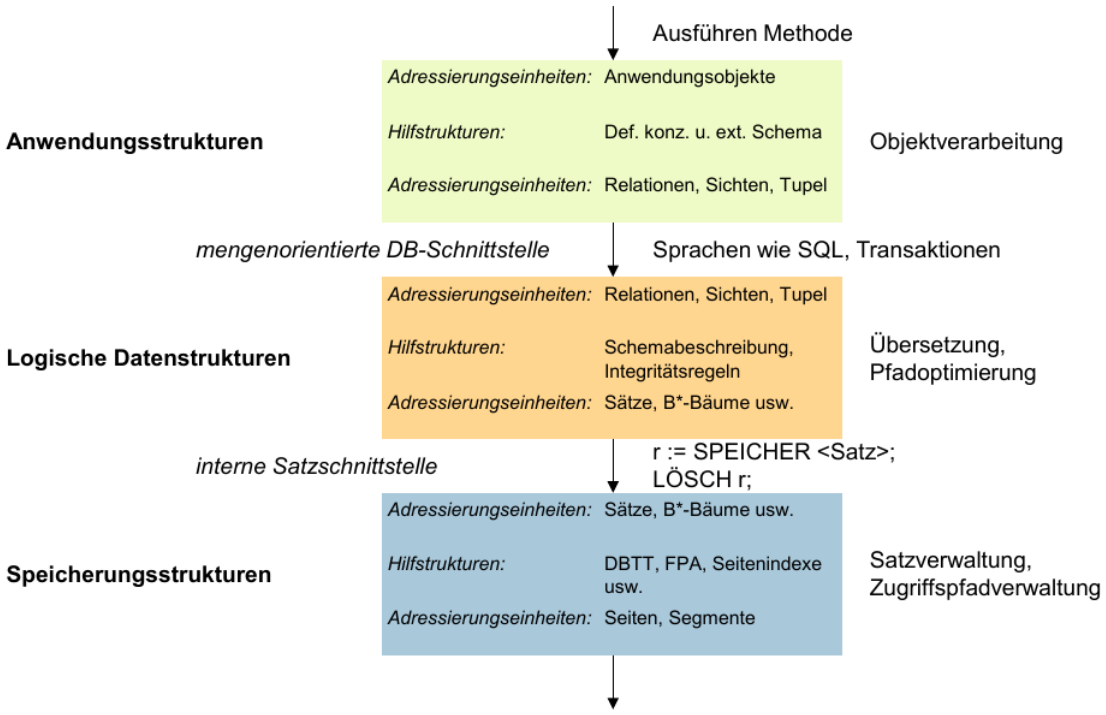

Abbildung 1: Schichtenübersicht

3. Was tut eine Schicht:
    - Realisiert einen bestimmten Dienst, den sie der Schnittstelle nach oben zur Verfügung stellt
    - Nimmt Dienste der darunter liegenden Schicht in Anspruch
    - Verbirgt darunter liegende Schicht vollständig und muss daher alle erforderlichen Funktionen anbieten
4. Was bedeutet Datenunabhängigkeit einer Anwendung:
   Speichern und Wiedergewinnen von persistenten Daten ohne Kenntnis der Details der Speicherung.

2 Sätze

2.1 Allgemein

Warum wurden Sätze eingeführt?
Sätze abstrahieren von Blöcken. Blöcke sind Hardware spezifisch. Sie haben unterschiedliche Längen und die Aufgabe von Einheiten von Platte und HS zu transportieren. Daher hat man eine anwendungspezifische Einheit eingeführt, die Daten zusammenfasst.

2.2 Satzadresse

Eine Satzadresse ist ein Bezeichner unter der man einen einzelnen Satz wiederfinden kann. Sie wird als stabil bezeichnet, wenn sich beim verschieben des Satzes die Adresse nicht ändert. → Neue Schicht über Blöcke

2.3 Wechselpuffertechnik

Optimierungsmöglichkeit bei sequenziellen Dateien. Man verwendet *zwei* Blockpuffer.
Anwendung beim

- Lesen: während Sätze in Puffer 1 gelesen werden **gleichzeitig** nächsten Block von Platte in Puffer 2 lesen
- Schreiben: wenn Puffer 1 voll, **asynchron** auf Platte schreiben und gleichzeitig Puffer 2 zur Aufnahme weiterer Sätze verwenden

2.4 Satzzugriff über Schlüsselwerte

2.5 Normales Hashing

Man möchte nun Sätze und deren Inhalt nicht nur über die Satzadresse wiederfinden, sondern auch über **inhaltliche Kriterien**. **Schlüssel** definieren ein oder mehrere Felder eines Satzes, über das man den Satz wiederfinden kann.
Lösung dafür ist **Hashing** der Sätze nach Schlüssel.
Kollisionen verhindern:

- Open Addressing (ausweichen auf Nachbar Buckets)
  - + kein zusätzlicher Speicherplatz erforderlich
  - − beim Suchen findet man im Bucket auch Überläufer
  - − beim löschen Überläufer zurückholen
  - − in Nachbarbuckets werden ggf weitere Überläufe erzeugt
- Overflow Buckets

  - Anlegen spezieller Überlauf Buckets für jeden Bucket
    * − zusätzlicher Speicherplatz erforderlich
    * + keine Mischung von sätzen
    * + keine Beeinträchtigung der Nachbarbuckets

2.5.1 Bewertung

- sehr schneller Zugriff über Schlüssel (1 bis 2 Blockzugriffe)
- gestreute Speicherung kann nur nach **einem** Schlüssel erfolgen
- Speicherplatz muss um voraus belegt werden

2.6 Virtuelles Hashing

Vorteile vom virtuellen Hashing ggü. normalen Hashing:
Das normale Hashing ist nicht erweiterbar, d.h. man muss den gesamten Speicher im voraus belegen. Dadurch kommt man in Platznot oder verschwendet zuviel. Beim linearen Hashing versucht man durch kontinuierliche Reorganisation Überlaufprobleme zu vermeiden.
Beschreibung von Dateizustand:

3 BBaum

Idee: Zusammenfassung ganz bestimmter Sätze in einem Block. Mehrwegbaum: jeder Knoten entspricht einem Block.

3.1 Aufbau

Am Anfang eines jeden Knotens steht n. n ist Anzahl der verwendeten Einträge, $k \leq n \leq 2k$ (in root $1 \leq n \leq 2k$).

Abbildung 2: BBaum Knoten

Danach folgen Tripel (Ki,Di,Pi) welche einen **Eintrag** bilden: Ki = Schlüsselwert
Di = Datensatz
Pi = Zeiger auf Nachfolgeknoten

3.2 Eigenschaften

- jeder Pfad ist perfekt balanciert
- jeder Knoten mit Ausnahme von root und Blättern hat mindesten $k + 1$ Nachfolger und höchstens $2k + 1$
- jeder Knoten (Ausnahme root) immer mindestens halb voll

3.3 Löschvorgang

1. Suche Knoten, in dem der zu löschende Schlüssel S liegt
2. Falls S in Blattknoten, dann lösche S und behandle ggf Unterlauf
3. Fall S in innerem Knoten dann untersuche linken und rechten Unterbaum von S
   - finde direkten Vorgänger S′ und Nachfolger S″
   - Wähle den aus der mehr Elemente hat
   - Ersetze zu löschenden Schlüssel S durch S′ oder S″ aus gewähltem Blattknoten und behandle ggf Unterlauf

3.4 Unterlauf

- Ein endgülter Unterlauf entsteht bei obigen Algorithmus erst auf Blattebene
- **Unterlaufbehandlung** wird durch Mischen des Unterlaufknotens mit seinem Nachbarknoten und darüber liegenden Diskriminator durchgeführt → Splitt rückwärts
- Unterlaufbehandlung endet in einem der Blätter!

3.5 B*-Baum

Alle Sätze werden in den Blattknoten abgelegt. Innere Knoten enthalten nur noch Verzweigungsinformationen, keine Daten. Am Ende eines Knotens ist ein Zeiger auf den nächsten enthalten.

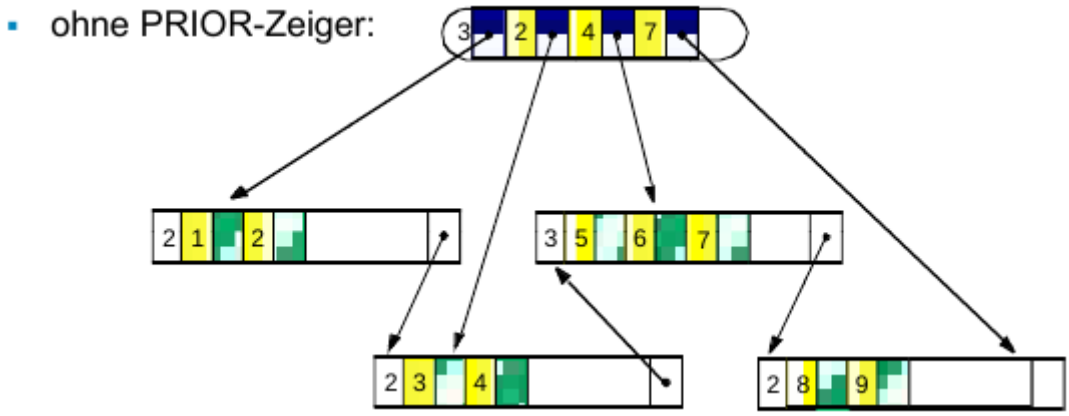

Abbildung 3: B*-Baum Aufbau

Merke:
Beim Löschen von Werten bleibt der gleiche Diskriminator in inneren Knoten enthalten.

3.5.1 Löschen

Entsteht durch das Löschen ein Unterlauf?

- Nein:
  Entferne Satz aus Blatt
- Ja:
  Mische das Blatt mit einem Nachbarknoten. Ist die Summe der Einträge in beiden Knoten größer als 2k?

  - Nein:
    Fass beide Blätter zu einem Blatt zusammen. Falls Unterlauf in Vaterknoten entsteht: Misch die inneren Knoten analog
  - Ja:
    Teil die Sätze neu auf beide Knoten auf, so dass ein Knoten jeweils die Hälfte der Sätze übernimmt. Der Diskriminator ist zu aktualisieren.

3.6 Vergleich

- BBaum:
  - keine Redundanz
  - lesen von Baum Inorder nur mit Stack von Höhe h
  - Gergingerer Verzweigungsgrad → größere Höhe
  - einige wenige Sätze (root) mit 1 Blockzugirff
- B*Baum:
  - Schlüsselwerte teilweise redundant
  - Kette von Blattknoten liefert alle Sätze nach Reihenfolge sortiert
  - hohe Verzweigung der inneren Knoten → geringe Höhe
  - für alle Blöcke müssen h Sätze gelesen werden
  - Schlüsselwerte der inneren Knoten müssen nicht in den Datensätzen vorkommen

3.7 Bitmap Index

B-Bäume (und Hashing) machen nur Sinn bei hoher Selektivität! ($\approx 5\%$) Lege für jeden Schlüsselwert eine Bitliste an. Bitwert 1 bedeutet, der Schlüssel hat im Satz den Wert zu dem die Liste gehört. 0 bedeutet er hat keinen anderen Wert.
Gut bei Wertigkeiten bis ca. 500. Hilft bei einfacher und effizienter Verknüpfung.

3.8 Primär- und Sekundär-Organisation

Primär-Organisation:
Bedeutung: Speicherung der Sätze selbst.
Sekundär-Organisation:
Bedeutung: verweist nur auf Sätze, die nach beliebigen anderen Kriterien abgespeichert werden. Ist allerdings nur möglich, wenn Primärorganisation Direktzugriff auf einzelnen Satz haben.
→ B-Baum/B*Baum als Sekundär Organsiation (Di); auch gestreute Speicherung als Sekundär Organsiation möglich (Buckets [Schlüsseltwer,Satzadresse] Paare)

4 Puffer

Frame = Im HS vorgesehener Platz zur Aufnahme eines Blocks → normalerweise so groß wie Block
Bei Zugriff auf Blöcke im HS müssen diese ggf eingelagert werden.

4.1 Ersetzungsstrategie

- wählt den zu verdrängenden Block aus
- Ziel: Minimierung von physischen Zugriffen

4.1.1 FIFO

Block der am längsten im Puffer ist wird ersetzt.
→ ungünstig da häufig benutzte Blöcke gerade im Puffer bleiben sollen

4.1.2 LFU (least frequently used)

Block auf den am seltensten zugegriffen wird ersetzt.

- − für sequentielles Lesen nicht brauchbar
- − hat ein Block einmal wieder zugegriffen bleibt der lange erhalten

4.1.3 LRU (least recently used)

- bewertetes Alter seit dem letzten Zugriff
- quasi verkettete Liste aller Blöcke im Puffer, wo bei Verdrängung letzter Block der Kette ersetzt wird

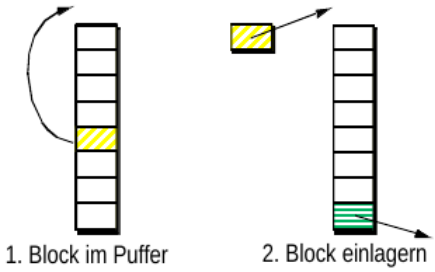

Abbildung 4: LRU

4.1.4 Clock

LRU Verhalten mit einfacher Implementierung. **Prinzip**:

- Benutzt(Dirty)-Bit eines Blocks im Puffer wird bei Zugriff auf 1 gesetzt
- bei Verdrängung zyklischer Suche mit dem Auswahlzeiger:
  - Falls Benutzt Bit = 1, wird es auf 0 gesetzt → Zeiger wandert zum nächsten Block
  - Falls Benutzt Bit = 0, Block wird ersetzt

4.2 Dienste einer Pufferverwaltung

Einkapselung der Pufferverwaltung:

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

- Block ist vor Verdrängung geschützt
- **Mode** gibt an welcher Block nur gelesen oder auch geändert werden soll

```
1 void Buffer::unix (char * BufferAddresse)
```

- gibt Block im Puffer zur Ersetzung frei

5 Programmschnittstellen

Vor- und Nachteile:
Programmzugriff:

- Keine SQL Injection nötig
- Kompakt
- spezieller Übersetzer nötig

Unterprogrammaufruf

- Optimierung und Typprüfung
- SQL Injection möglich
- Aufwändig für Programmierer

5.1 SQL Anfrage

Aufruf einer SQL Anfrage mit Index:

```
1 RecordFile index = new KeyedRecordFile("KNr","r");
2 TID tid = index.read.Key(23);
3 RecordFile saetze = new DirectRecord(Kunden", "r");
4 Record ergebnis = saetze.read(tid);
5 print(ergebnis.toString());
```

Aufruf einer SQL Anfrage ohne Index:

```
1 RecordFile saetze = new DirectRecordFile("Kunden","r");
2 Record ergebnis;
3 while (saetze.hasNext()){
4                 ergebnis = saetze.next();
5                 if(ergebnis.getKNr() == 23){
6                                 print(ergebnis);
7                                 break;
8                 }
9 }
```

5.2 SQL Schnittstelle

1. Verbindung aufbauen

```
1 Connection DriverManager.connect(username, password, database)
```

2. Methode um eine Anfrage auszuführen

```
1 Handle Connection.executeQuery (String query);
```

3. Methode um zu prüfen, ob es weitere Ergebnisse gibt

```
1 Boolean Handle.hasNext();
```

4. Methode um überhaupt an Ergebnisse zu kommen

```
1 Boolean Handle.hasNext();
```

6 Speicherung von Tupeln und Relationen

6.1 Ziele

- Relationen mit Mitteln der darunter implementierten Schichten sichern
- Anfragen (SQL) möglichst effizient auswerten

6.2 Speicherung von Tupeln in Sätzen

6.2.1 Aufbau von Sätzen

- Sätze sind aus **Feldern** zusammengesetzt (Name, Typ, Länge)
- **Systemkatalog**
  Informationen über Felder und Reihenfolge

6.2.2 Satztyp

- Menge von Sätzen mit gleicher Struktur → einmalige Beschreibung im Systemkatalog
- beim Speichern eines Satzes wird ihm ein Satztyp zugeordnet
- Länge der Sätze zumeist variabel

Annahme: Reihenfolge der Felder egal

6.2.3 Anforderungen

Speicherplatzeffizienz:

- variable Länge
- undefinierte Werte nicht speichern
- Hilfsstrukturen minimieren

direkter Zugriff auf Felder:

- ohne vorher andere Felder lesen zu müssen
- direkt zur Anfangs-Byte-Position innerhalb des Satzes

Flexibilität:

- Hinzufügen von Feldern bei allen Sätzen
- Löschen eines Feldes aus allen Sätzen

6.2.4 Speicherungsstruktur im Satz

Der Satz wird in zwei logische Teile untergliedert. Einer beinhaltet alle Felder fester Länge und der andere alle der variablen. Mit Katalogdaten und Längenangaben im Satz, lässt sich **flexibel** und **direkt** auf die Felder zugreifen.

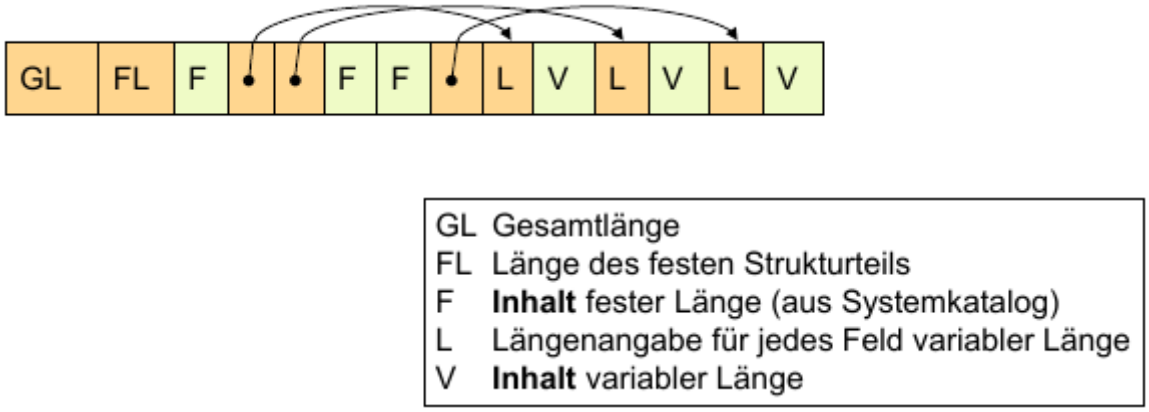

Abbildung 5: Variable Satz Speicherstruktur

Würden man alle Felder mit den Zeiger erreichen, wäre das Speicherplatzverschwendung. So sind alle Felder fester Größe direkt erreichbar.

Speicheurngsstruktur in Sätzen

Blatt 9 Aufgabe 2 üben.

6.3 Speicherung von Relationen

6.3.1 Spaltenweise Speicherung (C-Store)

- Auf das Lesen hin optimiert. Gut für Auswertung großer Datenmengen
- Nur Attribute lesen, die gebraucht werden
- kompakte Speicherung der Attributwerte

Prototyp

- speichert Sammlung von Spaltengruppen über mehrere Sätze verteilt
- Sortierschlüssel ist ein Attribut
- Gruppen von Spalten = Projektion
- Speicherung

  - **Schreibspeicher** für schnelles Einfügen und Ändern von Tupeln
  - **Lesespeicher** für umfangreiche Analysen
- Änderung durch Löschen und Einfügen durch Tuple Mover

Projektionen

- Die Projektionen besitzen ein oder mehrere Attribute aus einer logischen Tablle (ggf durch Fremdschlüssel auch Attribute aus anderen Tabellen)
- Duplikate bleiben erhalten
- Als Sortierschlüssel wird eines der Attribute hergenommen

Komprimierungen Abhängig von zwei Eigenschaften der Spalte:

- Sortierung (nach der Spalte selbst) : ja/nein
- Anzahl der verschiedenen Werte: wenig/viel

1. sortiert mit wenigen verschiedenen Werten:
   organisiert in einem B-Baum mit
   - dichter Packung → alle Seiten voll (keine Änderungen!)
   - und großen Seiten → geringe Höhe
2. unsortiert mit wenigen verschiedenen Werten:
   Offset-Indexe: B-Baum für Abbildung von Positiionen in einer Spalte auf die Werte in dieser Spaltebegin
3. sortiert mit vielen verschiedenen Werten:
   - B-Baum mit dichter Packung
   - Delta Codierung: Differenzen zum Vorgänger speichern
4. unsortiert mit vielen verschiedenen Werten
   - unkomprimiert
   - B-Baum mit dichter Packung als sek. Organisation möglich

6.3.2 Ausführung von Anfragen

Schritte für Ausführung von Query:

1. Syntaxprüfung
2. gibt es die Relation und Attribute
3. wie heißt Datei zur Relation
4. was für eine Datei ist das - direkt oder schlüsselbasiert
5. gibt es passende Projektionen
6. wo liegen die Verbund Indizes
7. müssen Partitionen zusammgeführt werden

7 Anfrageverarbeitung

7.1 Allgemein

Eine SQL Anfrage ist ein mengenorientierter Zugriff. D.h. diese mengenorientierten Operatoren müssen auf satzorientierte Operatoren abgebildet werden.

7.1.1 Phasen

Verarbeitungsschritte:

- Syntax überprüfen
- Rechte und Integritätsbedingungen (Formate) prüfen
- **Anfrageoptimierung** um effizient Query zu bearbeiten
- Code Generieung

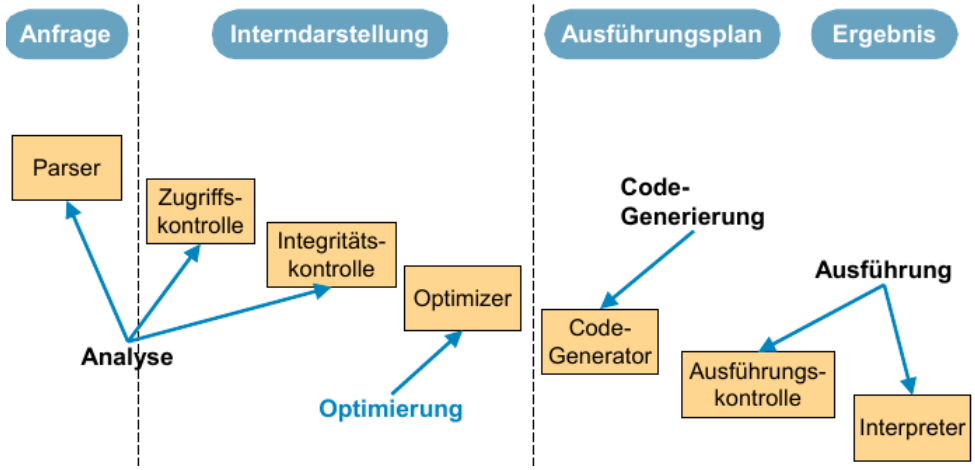

Abbildung 6: Phasen der Anfrageverarbeitung

Ziel:
Umsetzung deskriptiver Anfragen in eine „optimale“ Folge interner DBS-Operationen

7.2 Interndarstellung

7.2.1 Relationale Algebra

Definition von relationalen **logischen Operatoren**:

- Selektion: Auswahl von „Zeilen“ (**where**-Klausel)

- Projektion Auswahl von „Spalten“ (**select**-Klausel)
- Kreuzprodukt: Konkatenation derjenigen Tupel aus zwei Relationen(**from**)
- Verbund: Konkatenation derjenigen Tupel aus zwei Relationen die eine Bedingung erfüllen(**from-where**)
- Mengenoperationen auf zwei Relationen:
  - $R \cup S$ bzw **UNION(R,S)**
  - $R \cap S$ bzw **INTERSECT(R,S)**
  - $R \setminus S$ bzw **EXCEPT(R,S)**

7.2.2 Operatorbaum

Beispiel:

```
1  select Name, Beruf
2  from Abt a, Pers p,
3       PM pm , Projekt pj
4                    where a.ANr = p.ANr
5                    and a.AOrt = "Erlangen"
6                    and p.PNr = pm.PNr
7                    and pm.JNr = pj.JNr
8                    and pj.POrt = "Erlangen"
```

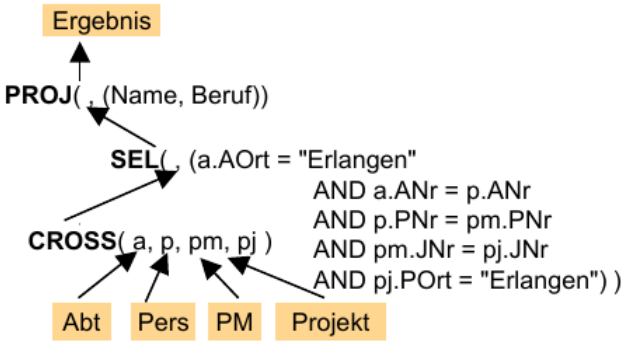

Abbildung 7: Operatorbaum

Blattknoten: Relationen
Gerichtete Kanten: Datenfluss
Knoten: Operatoren der Relationalen Algebra

7.3 Restrukturierung

Regeln:

1. Ein n-facher Verbund kann durch eine Folge von binären Verbunden ersetzt und werden und umgekehrt
2. Verbund ist kommutativ
3. Verbund ist assoziativ
4. Selektionen können zusammengefasst werden
5. Projektionen können zusammengefasst werden
6. Selektionen und Projektionen dürfen vertauscht werden
7. Selektionen und Verbund dürfen vertauscht werden
8. Selektion darf mit Vereinigung und Differenz vertauscht werden
9. Selektion und Kreuzprodukt können zu Verbund zusammengefasst weren

7.3.1 Algorithmus

Ziel: Kreuzprodukt vermeiden Heuristik:

- komplexe Verbundoperationen zerlegen in binäre Verbunde (Regel 1)
- Selektionen mit mehreren Prädikat Termen separieren in Selektionen mit jeweils einem Prädikat Term (Regel 4)
- Selektionen so früh wie möglich ausführen (Regel 7 und 8)
- Selektionen und Kreuzprodukt zu Verbund zusammenfassen, wenn das Selektionsprädikat Attribute aus den beiden Relationen verwendet (Regel 9)
- Projektionen so früh wie möglich ausführen, allerdings nicht vor Selektion und Mengenoperationen.

8 Relationale Operatoren

8.1 Allgemein

Die grundsätzliche Aufgabe besteht darin die logischen Operatoren (SEL(),PROJ(),JOIN(),...) durch **Planoperatoren** zu ersetzen.
Teilaufgaben:

- Erkennung gemeinsamer Teilbäume (notwendig: Zwischenspeicherung von Ergebnisrelation)
- Bestimmung der Verknüpfungsreihenfolge bei Verbundoperationen:
  - Ziel: minimale Kosten für die Operationsfolge
  - Heuristik: Minimierung der Größe der Zwischenergebnisse, d.h. kleinsten Relationen immer zuerst verknüpfen
- Gruppierung von direkt benachbarten Operatoren zu einem einzelnen Planoperator (z.B. Verbund mit Selektion)

8.2 Planoperatoren

SQL erlaubt Anfragen über k-Relationen:

- Ein Variablen Ausdrücke (eine Relation)
- Zwei Variablen Ausdrücke (zwei Relationen)
- k-Variablen Ausdrücke (zerlegt in Ein- und Zwei Variablen Ausdrücke)

8.2.1 Selektion

- Nutzung eines **Scan**-Operators
  Definition von Start- und Stoppbedingungen sowie Suchargumenten
- Index Scan → Auswahl des kostengünstigsten Index
- Relationen Scan

8.2.2 Projektion

- meist in Kobination mit Verbund, Selektion und Sortierung
- auch als eigener Planoperator

8.2.3 Sortierung

- erforderlich bei **order by**
- beschleunigt ggf Joins
- Duplikateleminierung (**distinct**)

8.2.4 Joins

Joins werden in **binäre** Verbunde gegliedert. Bei n Verbunden sind $n!$ Reihenfolgen möglich. Die optimale Reihenfolge ist abhängig von

- Planoperatoren
- passenden Sortierordnungen
- Größe der Operanden usw

Verbundoperationen sind sehr häufig und auch teuer → Optimierung.
Typisch sind **Gleichverbunde**. Standartszenario für Verbunde:

```
1 select * from R,S
2 where R.VA Θ S.VA  // Verbundpraedikat
3                    // Θ ∈ {=,>,<,≠,≥,≤}
4 and P(R.SA)        // lokale Selektion
5 and P(S.SA)        // VA = Verbundattribut
6                    // SA = Selektionsattribut
```

8.2.4.1 Nested Loop Verbund

Annahmen:
Sätze in R und S sind nicht nach Verbundsattributen geordnet.
Es sind keine Indexstrukturen $I_R(VA)$ und $I_S(VA)$ vorhanden.

```
1 Scan ueber S; //aeussere Schleife
2 fuer jeden Satz s, fuer den P(s.SA) gilt:
3     Scan ueber R; //innere Schleife
4     fuer jeden Satz r,
5             fuer den P(r.SA) AND (r.VA Θ s.VA) gilt:
6         uebernimm kombinierten Satz (r,s)
7         in das Ergebnis;
```

Komplexität: $\mathcal{O}(N^2)$

8.2.4.2 Sort-Merge Verbund

Zweiphasiger Algorithmus:

Phasen:

1. Sortierung von R und S nach R.VA und S.VA, dabei Eliminierung nicht benötigter Tupel (durch Prüfung von P(R.SA) oder P(S.SA))

2. schritthaltende Scans über R- und S-Relationen mit Durchführung des Verbundes bei r.VA=s.VA

Komplexität: $\mathcal{O}(N \log N)$

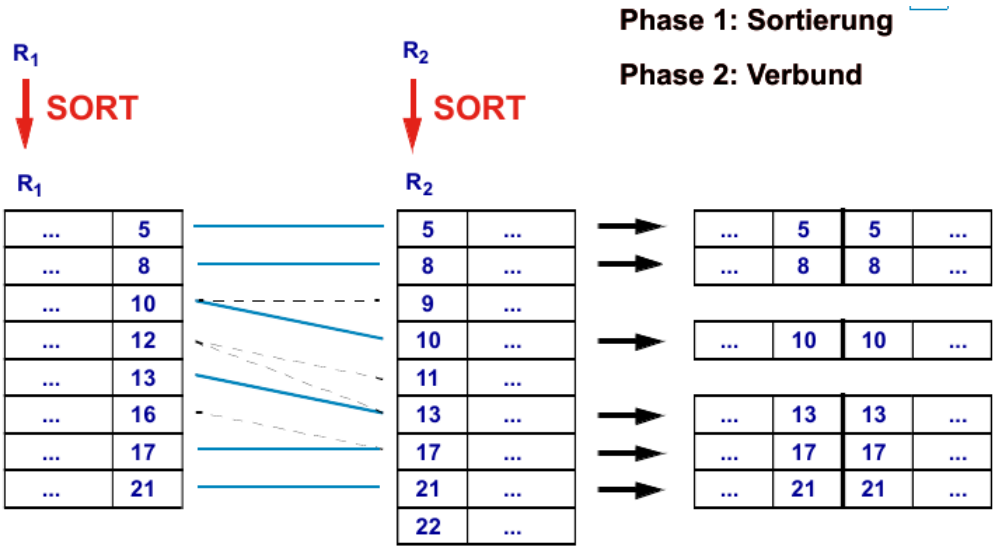

Abbildung 8: Merge Sort Verbund

8.2.4.3 Hash Verbund

Spezialisierung für **Gleichverbund**. Da der Hauptspeicher immer größer ist, lassen sich Zwischenergebnisse besser ausnutzen.
Idee:

- Tupel der einen Relation im HS ablegen, so dass sie über VerbundAttribut schnell gefunden werden können
- Tupel der anderen Relation sequenziell durchlaufen und mit Wert des VerbundAttributs die passenden Verbundpartner im HS aufsuchen
- Organisation der Tupel im HS über **Hashing**

Einfachster Fall: **Classic Hashing**:
Funktionsweise:
Äußere Schleife

- Abschnittweise lesen der kleineren Relation R

- Relation wird in p Abschnitte aufgeteilt, die alle in HS passen
- Aufbau einer Hash Tabelle mit $h_A(r.VA)$

Innere Schleife

- Überprüfung für jeden Satz von S mit P(S.SA) → ebenfalls mit Hashing
- wenn sich Verbundpartner in dieser Adresse befindet, Durchführung des Verbundes

Komplexität: $\mathcal{O}(p \times N)$. Da Verbundpartner S p-mal gelesen werden muss.

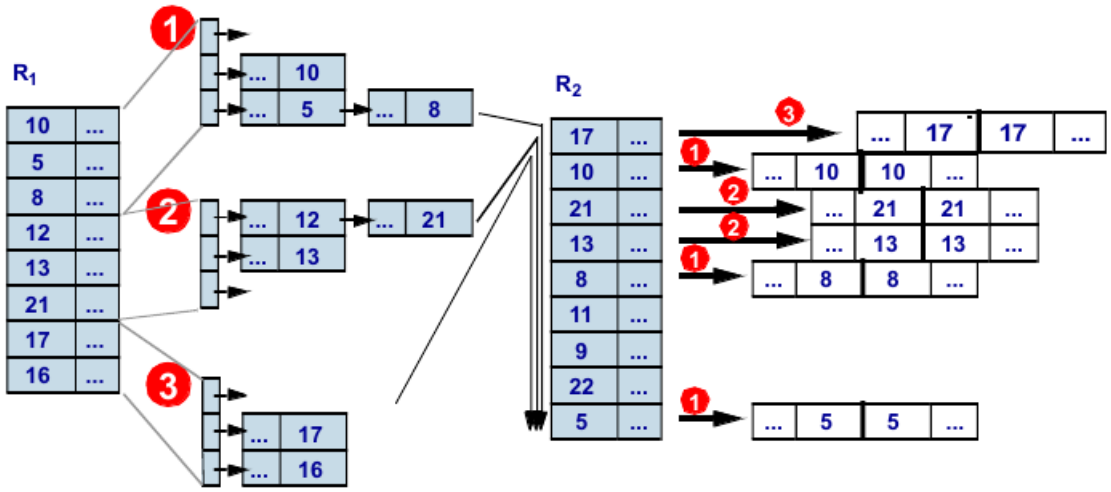

Abbildung 9: Hash Verbund

Problem ist, dass S p-mal durchlaufen werden muss. Daher die Idee S analog zu R zu partitionieren.
Stichwort Simple Hashing.

8.3 Anfrageoptimierung

Ziel:
→ Ermittlung des *kostengünstigsten* Auswertungsweges
Zentrales Problem:

- globales Optimieren hat hohe Komplexität (NP-Schwer zur Laufzeit)
- Einsatz von Heuristiken, da nicht alles nötige Wissen immer vorhanden ist

Optimierungsziel:

- Maximierung des Outputs bei gegeben Ressoucren

- Minimierung der Resourcennutzung

Die Wichtigsten Kostenarten sind:

- Berechnungskosten (CPU)
- I/O Kosten (Anzahl physischer Referenzen)

8.3.1 Ausführungsplan

Ziel ist es eine möglichst guten Ausführungsbaum zu erstellen. Problematisch ist die riesige Anzahl an Möglichkeiten mit steigender Komplexität der Anfrage (z.B. Query mit 15 Verbunden 10^{70} Möglichkeiten).
Diese Vielfalt entsteht durch verschiedene Implementierungen der Planoperatoren und der Operationsreihenfolgen.
→ **Ziel der Plangenerierung**:

- finden von Plänen gelingt immer schnell
- mit möglichst wenig generierten Plänen auskommen

Unterschiedliche Strategieklassen:

- voll enumerativ
- beschränkt enumerativ
- zufallsgesteuert

8.3.1.1 Kostenformeln

- gewichtetes Maß für I/O und CPU Belastung:
  C = #physische-Seitenzugriffe + W × #Aufrufe-des Zugriffsystems
- CPU-bound : höherer I/O-, geringerer CPU Aufwand:
  W_{CPU} = #Instr-pro-Aufruf-des-Zugriffsystems /
  #Instr-pro-I/O-Vorgang
- I/O-bound : geringere I/O-, höherer CPU Aufwand:
  W_{IO} = #Instr-pro-Aufruf-des-Zugriffsystems /
  (#Instr-pro-I/O-Vorgang + Zugriffzeit × MIPS-Ratte)

8.3.1.2 Selektivitätsabschätzung

Der **Selektivitätsfaktor** beschreibt den erwarteten Anteil an Tupeln, die ein Prädikat p erfüllen.
Diese Trefferate gibt auch an, inwiefern eine vorhandene Indexstruktur die Laufzeit reduzieren.

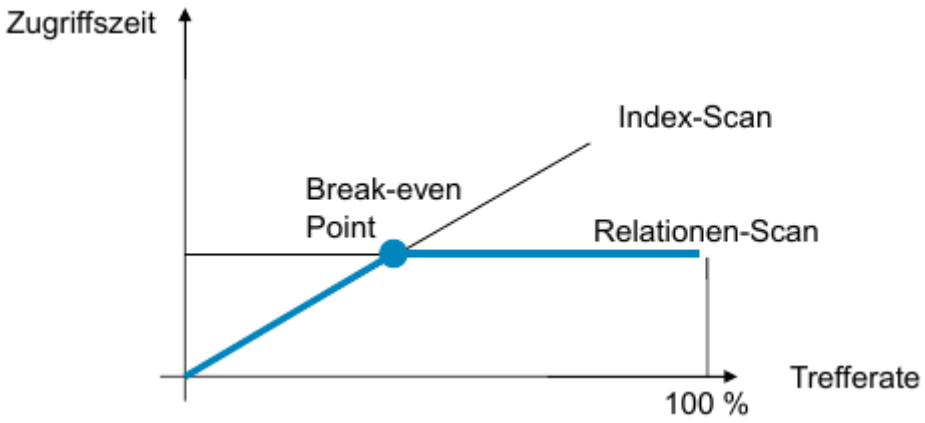

Abbildung 10: Qualitatives Zugriffsdiagramm

Nur bei sehr geringen Trefferraten lohnt sich ein Index Scan!

9 Transaktionen

9.1 Allgemein

9.1.1 Was sind Transaktionen?

Transaktionen sind eine logische Arbeitseinheit, die mehrere DB Schritte zusammenfassen. Z.B. einfügen eines neuen Mitarbeiters besteht aus vielen kleinen DB Befehlen, bis Datenbank wieder konsistent ist

9.1.2 Wofür sind sie gut?

Sie überleiten von einem konsistenten Zustand in den nächsten. Im Fehlerfall muss die Möglichkeit gegeben werden, wieder in einen konsistenten Zustand zu finden. Im Mehrfachbetrieb muss gewährleistet werden, dass keine Nebenläufigkeitsprobleme auftauchen.

9.1.3 ACID Eigenschaften von Transaktionen:

- **Atomic:** Alle Änderungen von TA werden aktiv oder keine
- **Consistency:** TA überführt DB von konsistent in konsistenten Zustand
- **Isolation:** TA wird nicht von anderer TA beeinflusst
- **Durability:** Änderungen sind Dauerhaft in DB übernommen

9.2 Konsistenzen:

9.2.1 Physische Konsistenz:

Alle Speicherungsstrukturen sind korrekt. Alle TID's stimmen etc.

9.2.2 Logische Konsistenz:

Setzt Physische Konsistenz voraus.
Korrektheit der Dateninhalte.
Alle Bedingungen des Datenmodells und alle Benutzerdefinierten Bedingungen sind erfüllt (Assertions).

9.3 Synchronisation

9.3.1 Zustandsdiagramm TA

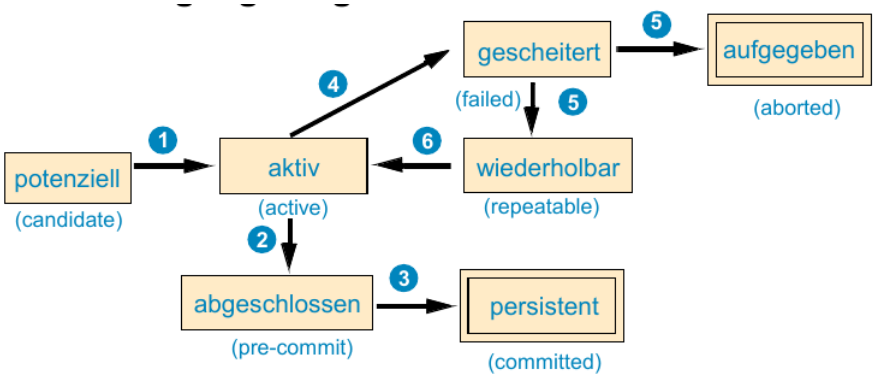

Abbildung 11: Transaktionszustände

9.3.2 Anomalien ohne Synchronisation

- **lost updates**
  $r_1(x)r_2(x)w_1(x)w_2(x)$
- Abhängigkeiten von nicht freigegeben Änderungen
  - **dirty read**
    $w_1(x)r_2(x)a_1c_2$
    Änderung wird noch nicht freigegeben (ggf abgebrochen) aber trotzdem gelesen
  - **dirty write**
    $w_1(x)r_2(x)w_2(x)a_1c_2$
    Änderung wird noch nicht freigegeben (ggf abgebrochen) aber trotzdem geschrieben
- **Non Repeatable Read**
  $r_1(x)w_2(x)r_1(x)$
  T1 sieht beim erneuten lesen einen anderen Wert als zuvor
- **Phantom**
  $r_1(P)w_2(xinP)r_1(P)$
  Wie bei *Non Repeatable Read* nur Prädikat statt einzelnes Element

Legende: c = *commit,* a = *abort,* w = *write,* r = *read*

9.3.3 Serialisierung

Hintereinanderausführung aller TA's. Problem werden sehr große Wartezeiten. Allgemein ist deine Datenbank für Mehrbenutzerbetrieb ausgelegt, dies würde ad adsurdum geführt durch Serialisierung. Was tun im Fehlerfall

9.3.4 Serialisierbarkeit

Ziel:
TA's laufen *verzahnt* ab, aber ihr Ergebnis gleicht dem eines seriellen Ablaufes.
→ Ein Schedule von TA's ist **serialisierbar**, wenn es einen äquivalten **seriellen** Ablauf gibt.

9.3.5 Implementierung von Sperren

Wann sperren?
Statisch:
Alles sperren was evtl gebraucht wir (preclaiming)
Dynamisch:
Zur Laufzeit von TA nach Bedarf sperren
→ Gefahr von Verklemmungen!
Sperrgranulat:
Warum nicht Tupel:

- Nicht immer effizient, bei großen Mengen
- Phantom Tupel; es können nur bereits existente Tupel gesperrt werden.

Deshalb wird *hierarchische Schachtelung* der Datenobjekte:

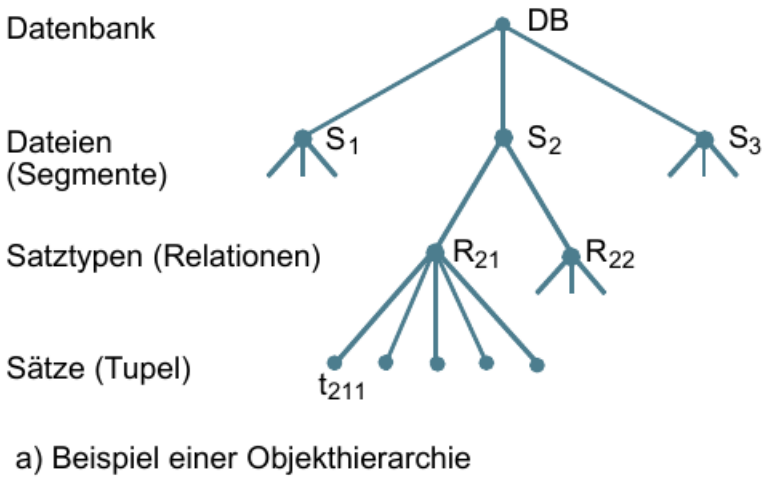

Abbildung 12: Sperrgranulate

Für jedes Objekt kann man nun einen Sperrmodus vergeben. Andere TA können daran erkennen, ob sie ihre Aktion ausführen dürfen.

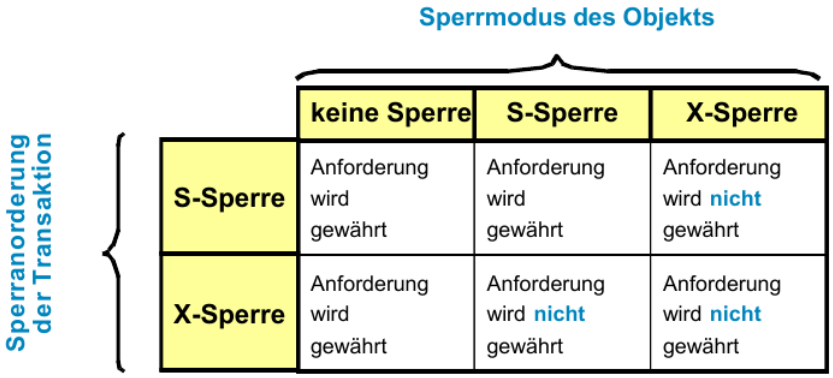

Abbildung 13: Kompatibilitätsmatrix

Das Problem an dieser Art der Sperrung ist, dass

- Alle Nachfolgeknoten implizit mitgesperrt werden
- Alle Vorgängerknoten auch gesperrt werden müssen

Daher werden sogenannte **(Intention) Anwartschaftssperren** eingeführt:

- IS Sperre:
  falls auf untergeordnete Objekte lesend zugegriffen wird

- IX Sperre:
  falls auf untergeordnete schreibend zugegriffen wird

→ Benutzung von Untersperre wird angezeigt, aber nochmal explizit weiter unten gesetzt!

9.3.5.1 TOP-DOWN bei Sperre

- Bevor ein Knoten mit S oder IS gesperrt werden darf, müssen alle Vorgänger in der Hierarchie im IX- oder im IS-Modus gesperrt worden sein.
- Bevor ein Knoten mit X oder IX gesperrt werden darf, müssen alle Vorgänger in der Hierarchie im IX-Modus gesperrt worden sein.

9.3.5.2 BOTTOM-UP bei Freigabe

- Freigabe von unten nach oben
- kein Knoten darf entsperrt werden, wenn noch andere Nachfolger dieses Knotens gesperrt sind

Als Optimierung gibt es noch die **SIX** Sperre. Alles wird lesend gesperrt und eine Menge an Nachfolgern schreibend. Das lohnt sich z.B. wenn aus einer Relation nur wenige Tupel verändert werden.

9.3.6 Probleme beim Sperren

- Sperren muss sehr schnell gehen, da Anforderungen sehr hoch
- halten von Sperren bis TA-ende führt zu langen Wartezeiten
- Eigenschaften des Schemas können „hotspots“erzeugen
- Optimierungen:
  - Nutzung mehrere Objektversionen
  - spezialisiertes Sperren
  - Änderungen auf privaten Objektkopien

9.3.7 Verklemmungen

Lösungsmöglichkeiten:

- **Timout:**
  Transaktion nach bestimmter Wartezeit zurücksetzen.
  → problematisch Länge des Timeouts zu bestimmen
- **Verhütung (Prevention)**
  Durch Preclaiming (s.o.) keine Deadlock Verhinderung zur Laufzeit notwenig.

- **Vermeidung (Avoidance)**
  Potentielle Deadlocks im vorhinein erkennen und vermeiden
  → Laufzeitunterstützung notwendig
- **Erkennung (Detection)**
  Explizites führen eines Wartegraphen und darin Zyklensuche. Im Zyklus dann eine oder mehrere (am besten billigste) TA zurücksetzen.

Deadlock - nicht serialisierbarer Schedule

Nicht serialisierbare Schedules, beschreibt Abläufe die zu keinem seriellen Ablauf äquivalent sind. Die somit zu Deadlocks führen. Führt man diese mit *dynamischen* Sperren aus, muss es aber nicht zwangsläufig auch zu Deadlocks kommen!

9.4 Recovery

Recovery Klassen:

- Partial Undo (R1-Recover)
  - nach Transaktionsfehler
  - isoliertes und vollständiges Zurücksetzen der Daten in Anfangszustand
  - beeinflusst andere TA nicht
- Partial Redo (R2-Recover)
  - nach Systemfehler (mit Verlust von HS)
  - Wiederhlung aller verlorengegangenen Änderungen (waren nur im Puffer) von abgeschlossenen TA
- Global Undo (R3-Recover)
  - nach Systemfehler (mit Verlust von HS)
  - Zurücksetzen aller durch Ausfall unterbrochenen TA
- Global Redo (R4-Recover)
  - nach Gerätefehler
  - Einspielen von Archivkopie und nachvollziehen aller beendeten TA's

9.4.1 Archivkopien

Arten:

- „cold backup“: DBS muss außer Betrieb sein
- „hot backup“: Beeinträchtigung des laufenden Betriebs

9.4.2 Einbringstrategien

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

- STEAL:
  Bei Verdrängung aus dem Puffer, auch vor Ende der Transaktion
- NO STEAL:
  Frühestens am Ende der Transaktion → kein UNDO erfoderlich (aber großen Puffer)
- FORCE:
  Spätestens am Ende der Transaktion → kein Partial Redo erfoderlich
- NO FORCE:
  Erst bei Verdrängung aus dem Puffer

Wie werden geänderte Daten aus dem Puffer auf Platte geschrieben?

- ATOMIC:
  Indirekte Einbringstrategie.
  Ununterbrechbares Umschalten
- NOT ATOMIC: Direkte Einbringstrategie.
  Ist nicht ununterbrechbar.

9.4.3 Protokolldaten

Als Protollinformation zählt was über die Einbringungsstrategie hinaus benötigt wird, um nach einem Systemausfall den jüngsten konsisten Zustand wiederherzustellen.

In welcher Form?	Was?	
	Zustände	Übergänge
logisch	?	Änderungs-operationen
physisch	Before-Images, After-Images	EXOR-Differenzen

Abbildung 14: Protokollinformationen

9.4.3.1 Wann wird in Protokolldatei geschrieben?

- UNDO-Information:
  - bevor die zugehörigen Änderungen in Datenbestand eingebracht werden
  - sonst Rücksetzen unmöglich

- REDO-Information:
  - muss geschrieben sein (in tmp Log Datei und Archivdatei), bevor der Abschluss der TA an Programm bzw Benutzer gemeldet wird
  - sonst Wiederherstellung nicht möglich

9.4.4 Backward-/Forward Recovery

1. Forward:
   Änderungen sind noch nicht in DB, aber schon in Log. Bevor diese auf DB geschrieben werden, wird ein Commit-Record in Log File geschrieben. Falls was schief geht und der Commit Record noch da ist, werden alle diese Änderungen wiederholt und ansonsten nichts getan, da alte Version noch in DB steht.

2. Backward:
   Ins Log kommt immer **alte** Version eines Wertes. Bei einem Commit wird gewartet bis alle neuen werte in DB stehen. Danach erst wird CommitRecord erstellt und eine Bestätigung signalisiert. Bei Verlust werden entweder alte Daten hergestellt oder nichts getan.

9.4.4.1 Vergleich

- Forward Recovery: Hoher Speicherplatzbedarf! Daten werden erst nach Beendigung der TA in DB geschrieben und deswegen solange in Puffer behalten
- Backward-Recovery: Hoher I/O Aufwand! Alle Änderungen müssen vor TA-Ende in Datenfiles stehen.

9.4.5 Undo-/Redo Logging

Verbesserung zu Backward-/Forward Recovery.

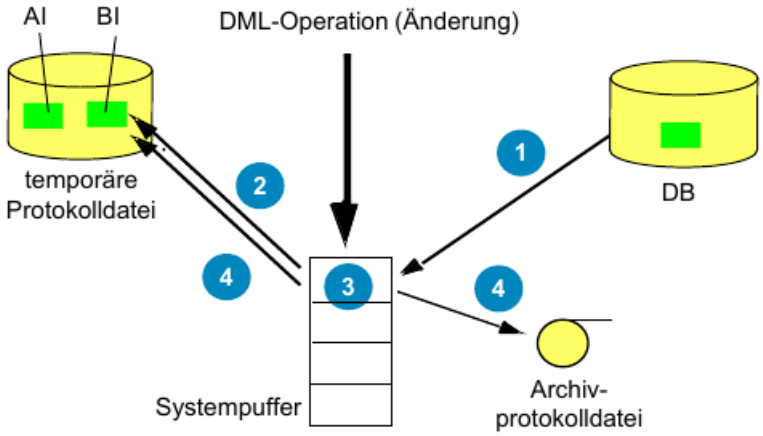

Abbildung 15: Physische Protokollierung

Zustände vor bzw nach einer Änderungen werden protokolliert:

- alter Zustand: Before Image (BI) für UNDO
- neuer Zustand: After Image (AI) für REDO
- für jede veränderte Seite (3) wird jeweils eine vollständige Kopie vor (2) und nach (4) der Änderung in den Log geschrieben.
  + schnelle Recovery
  − hoher E/A Aufwand

9.4.5.1 Optimierungen

Um Log-Aufwand zu reduzieren nur geänderte **Teile einer Seite** protokollieren.
Sammlung und Pufferung im HS:

- + reduzierter E/A Aufwand
- − komplexeres und zeitaufwändigeres Recovery
  Zurückspeichern gleicht eher dem Neu einfügen (freien Platz suchen, TID vergeben), da Stelle von wo Einträge stammen anderweitig genutzt werden können

9.4.6 Sicherungspunkte

Maßnahmen zur Begrenzung des REDO Aufwands nach Systemfehlern (alle erfolgreichen Änderungen die im Puffer verloren gegangen sind müssen wiederholt werden → nicht praktikabel!

Methoden

- **Transaction-Oriented Checkpoint**
  Geänderte Seiten einer TA nach TA-ende sofort in DB bringen (siehe FORCE) → zu hohe Belastung
- **Transaction-Consisten Checkpoint**
  - Einbringung aller Änderung erfolgreicher TA's
  - Lesesperre auf ganzer DB zum Zeitpunkt des Sicherungspunktes
  - Verzögerung von TA's
  - Sicherungspunkt begrenzt Undo und Redo Recovery
- **Action-Consisten Checkpoint**
  - Zum Zeitpunkt des Sicherungspunktes dürfen keine Änderunen aktiv sein
  - Begünstigt nur Recovery, dafür kürzere Totzeit von System

9.4.7 Allgemeine Restart Prozedur

3-phasiger Ansatz

1. Analyse Lauf
   Vom letzten Checkpoint bis zum Log Ende. Bestimmung von Gewinner und Verlierer TA's, sowie Seiten, die von ihnen geändert wurden
2. Undo Lauf
   Rücksetzen der Verlierer TA's durch Rückwärtslesen des Logs bis zum BOT Satz der ältesten Verlierer TA
3. Redo Lauf
   Vorwärtslesen des Logs (Startpunkt abhängig von Checkpoint Typ) und Änderungen der Gewinner TA's wiederholen