Zusammenfassung IDB

Marco Ammon

12. Februar 2019

1 Einführung

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

2 Dateiverwaltung

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

3 Sätze

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

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

4 Schlüssel

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

        · betrachte doe Blattknoten mit direktem Vorgänger S' und direktem Nachfolger S'' von S
        · wähle den größeren
        · ersetze S je nach Wahl durch S' bzw. S''
        · lösche entsprechenden Schlüssel S' bzw. S'' und ggf. Unterlauf behandeln
  - *B*-Baum* / *B+-Baum*:
    * Sätze stehen *ausschließlich* in Blattknoten
    * innerer Knoten:
      · Anzahl der verwendeten Einträge n
      · Eintrag: Tupel (Referenzschlüssel, Blocknummer des Kindknotens)
    * Blattknoten:
      · Anzahl der verwendeten Einträge n
      · Vorgänger-Zeiger, Nachfolger-Zeiger
      · Eintrag: Tupel (Schlüsselwert, Datensatz)
    * Löschen ohne Unterlauf: lösche Satz aus Blatt; Diskriminator muss *nicht* geändert werden
    * Löschen mit Unterlauf:
      · Ist Anzahl der Einträge des Blatts und eines Nachbarknotens größer als 2k, verteile Sätze neu auf beide Knoten
      · ansonsten mische beide Blätter zu einem einzigen
  - Müssen nicht zwangsläufig zur *Primärorganisation* verwendet werden, können als „Sätze“ z.B. auch nur Satzadressen enthalten

- *Bitmap-Indizes*: eine Bitmap *pro Schlüsselwert*

5 Puffer

- Hauptspeicherbereich, der Blöcke aufnehmen kann, um (Lese-/Schreibe-) Zugriffe zu *beschleunigen*

- *Ersetzungsstrategie*: „Welcher Block wird verdrängt?“
  - *first in, first out* (FIFO): „ältester“ Block
  - *least frequently used* (LFU): am seltensten benutzter Block
  - *least recently used* (LRU): am längsten nicht mehr benutzter Block
  - *second chance* (CLOCK): Approximation von LRU mit einfacherer Implementierung:
    * Jeder Block im Puffer besitzt ein *Benutzt-Bit*
    * bei Verdrängung Suche mit Zeiger
    * falls Benutzt-Bit 1, auf 0 setzen
    * falls Benutzt-Bit 0, Block ersetzen
    * **TODO:** Muss immer weitergegangen werden?

- Zustand im Fehlerfall hängt unter anderem von *Einbringstrategie* (siehe **TODO** Recovery) und *Seitenzuordnung* ab

- Seitenzuordnung: „Welche Blöcke (in einer Datei) gehören zu einer Seite (im Puffer)?“
  - *direkt*: aufeinander folgende Seiten werden auf aufeinander folgende Blöcke einer Datei abgebildet
  - *indirekt*: *Page Table* enthält zu jeder Seite eine Blocknummer

- Seiteneinbringung:
  - *direkt*: Bei Verdrängung aus Puffer wird genau der Block überschrieben, aus dem ursprünglich eingelagert wurde („update-in-place“)
  - *indirekt*: Bei Verdrängung aus Puffer wird in einen freien Block geschrieben.
- Problem der indirekten Seiteneinbringung: „Wann können alte Blöcke gelöscht werden?“; verschiedene Lösungsansätze:
  - *Schattenspeicher*:
    - Änderungen nur auf Kopien, die periodisch dann mit „gesicherter“ Version vertauscht wird
  - *Twin Slots*:
    - jede Seite hat zwei Blöcke
    - immer beide lesen, bei Änderungen älteren überschreiben

6 Programmzugriff

- *Precompiler* übersetzt SQL-Anweisungen (mittels `EXEC SQL` gekennzeichnet) zur *Compiler-Zeit* in die verwendete Programmiersprache
  - Deklaration der verwendeten Variablen am Anfang mittels `DECLARE SECTION`
  - Fehlermeldungen und ähnliches werden über die sogenannte *SQL communication area* verwaltet (`INCLUDE SQLCA` am Anfang)
  - Mengen-orientes Paradigma des DBVS oft nicht mit Programmiersprache vereinbar ⇒ Einrichtung eines *Cursors* zum tupelweisen Durchlaufen der Ergebnismenge
  - Beispielablauf: `DECLARE CURSOR` → `OPEN` → (mehrfach) `FETCH` bis Fehlercode == 100 → `CLOSE`
  - kann durch Präprozessort direkt als *stored procedure* **TODO: Verlinkung** angelegt werden
- *Unterprogrammaufruf* (*Call-Level-Interface*):
  - Übergabe der SQL-Anweisungen zur *Laufzeit*
  - Beispiel JDBC
    - `Connection con = DriverManager.getConnection(URL, USER, PASSWORD);`
    - `Statement anweisung = con.createStatement();`
      `ResultSet ergebnis = anweisung.execureQuery(ANFRAGE);`
      außerdem: `int executeUpdate(String sql)`, `boolean execute(String sql)`
    - `while (ergebnis.next()) {`
      `int pnr = ergebnis.getInt(1);`
      `}`
  - Bei mehrfacher Ausführung der gleichen Abfrage mit unterschiedlichen Werten *prepared statements* sinnvoll:
    - Anfrage enthält Platzhalter für Werte
    - Analyse, Ausführungsplanerstellung und weiteres wird sofort durchgeführte
    - JDBC:
      - `PreparedStatement prep = con.prepareStatement(ÏNSERT ...VALUES (?,?))`
      - Setzen der Werte mittels `void setDATENTYP(int paramId, DATENTYP val)`
      - Ausführung mit `prep.executeUpdate()`
  - bei stored-procedures nur noch einmaliges Analysieren, etc. zur Compile-Zeit erforderlich:
    - Prozedur in DBVS bekommt „Namen“, über den sie mit Werten als Parametern aufrufbar ist

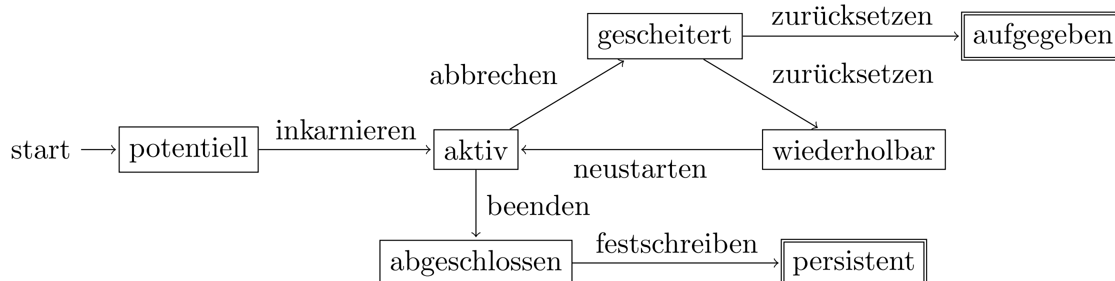
    * JDBC:
        · `CallableStatement call = con.prepareCall("{ call PROZEDUR }");`
        · Eingabe-Parameter analog zu prepared statements
        · Ausgabe-Parameter mittels `registerOutParameter(int paramId, int type)`

- *O/R-Mapping* bildet Objekte der Programmiersprace (meist durch Annotationen) auf Tupel der relationalen DB ab

7 Transaktionen

- sinnvoll für *nebenläufigen* Zugriff
- erleichtern Umgang mit *Fehlern* und Ausfällen
- *Programmfehler*: Absturz des Datenbank-Anwendungsprogramms ⇒ Daten im Puffer und auf Festplatte in undefiniertem Zustand
- *Systemfehler*: DBVS oder BS fällt aus, Hardware-Fehler, . . . ⇒ Daten im Puffer verloren, auf Festplatte in undefiniertem Zustand
- *Gerätefehler*: Festplattenausfall ⇒ Daten auf Festplatte sind verloren
- *physische Konsistenz*:
  - Korrektheit der Speicherungsstrukturen, Verweise und Adressen
  - alle Indizes sind vollständig und stimmen mit Primärdaten überein
- *logische Konsistenz*:
  - Korrektheit der Inhalte
  - Referentielle Integrität, Primärschlüsseleigenschaft und eigene Assertions sind erfüllt
  - erfordert physische Konsistenz
- Nach Fehler soll ein logisch konsistenter Zustand erreicht werden:
  - der Zustand vor Beginn der unvollständigen Änderungen durch *Rückgängigmachen* dieser (*undo*)
  - der Zustand nach Abschluss aller Änderungen durch *Vervollständigung* bzw. *Wiederholung* der unvollständigen Änderungen (*redo*)
- *Sicherung* und *Protokollierung* *immer* notwendig
- *Transaktion* als *logische Einheit* einer Folge von DB-Operationen (von einem logisch konsistenten Zustand zum nächsten):
  - bei Fehler vor Ende: Rückgängigmachen der bisher durchgeführten Änderungen
  - bei Fehler nach Ende: kein Problem
  - Anfang meist implizit (oder *begin*)
  - Ende durch commit (Änderungen sollen durchgeführt werden) bzw. *abort* (Änderungen sollen verworfen werden)
- *ACID*-Eigenschaften einer Transaktion:
  - *Atomarität* („alles oder nichts“ wird ausgeführt)
  - *Konsistenz*
  - *Isolation* (gegenüber anderen Zugriffen auf DB)
  - *Dauerhaftigkeit* (auch nach Fehler bleiben erfolgreiche Transaktionen bestehen)

- „Lebenszyklus“ einer Transaktion



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

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

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

8 Speicherung

- Speicherung der Tupel in Sätzen:
  - zusammengesetzt aus Feldern mit Namen, Typ und Länge (maximal oder variabel)
  - *Metadaten* in *Systemkatalog* gespeichert
  - *Satztyp*: Menge von Sätzen gleicher Struktur (z.B. Tupel einer Relation)

- verschiedene *Speicherungsstrukturen* in Sätzen:
  - mit *eingebetteten Längenfeldern*: Gesamtlänge, Inhalt fester Länge, zu jedem Inhalt variabler Länge vorher die Länge → satzinterne Adresse kann *nicht* direkt aus Katalogdaten berechnet werden
  - eingebettete Längenfelder mit *Zeigern*: Gesamtlänge, Länge des festen Teils, Inhalte fester Länge, Zeiger auf Längenangabe variabler Felder, variable Felder → satzinterne Adresse kann aus Katalogdaten berechnet werden **TODO:** Grafik
- *spaltenweises* Abspeichern mittels *C-Store*:
  - vor allem auf das Lesen optimiert
  - Änderungen durch Löschen und Einfügen
  - *Projektion*:
    * eine oder mehrere Spalten einer Tabelle (und ggf. anderer, über Fremdschlüssel erreichbare Tabellen) nach einem Attribut sortiert
    * Speicherschlüssel (*Storage Keys*, *SK*) für jedes Tupel, aus Position berechenbar
    * *Verbund-Indizes* (*Join Indices*): Seien T1 und T2 Projektionen der Tabelle T, dann ist ein Join-Index von T1 zu T2 eine Liste von Tupeln aus T2 um den jeweiligen SK aus T1 ergänzt.
  - verschiedene *Komprimierungen* abhängig von Sortierung und Anzahl der verschiedenen Werte:
    * sortiert, wenige verschiedene Werte: Tripel (Wert, Position des ersten Auftretens, Anzahl des gleichen Werts) in B-Baum
    * unsortiert, wenige verschiedene Werte: lauflängenkodierte Bitmaps pro Wert mit B-Baum zum Auffinden der richtigen Bitmap
    * sortiert, viele verschiedene Werte: Delta-Kodierung (DIffertenz zum Vorgänger) mit B-Baum als Primärorganisation
    * unsortiert, viele verschiedene Werte: unkomprimiert, bei Zeichenketten *Dictionary*

9 Anfrageverarbeitung

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

- *Schnitt* INTERSECT(R,S)
- *Ausschluss* EXCEPT(R,S)
- analoge Operationen auf *Multimengen*
- *Umbenennung* RENAME$(R, R_{\text{neu}}, ((A_i, A_{i,\text{neu}}), \dots))$
- *Duplikat-Eliminierung* DUP-ELIM(R)
- *Aggregation* SUM(R, A_i), AVG(R, A_i), MIN(R, A_i), MAX(R, A_i), COUNT(R)
- *Gruppierung* GROUP(R, L, G) mit $G = ((\text{AGG}_1, (A_i), \text{name}_1), \dots)$
- *erweiterte Projektion* G-PROJ(R, L) mit $L = (\text{name}_1 = \text{expr}_1, \dots)$
- *Sortierung* SORT(R, L) mit $L = (A_i, A_j, \dots)$
- *äußerer Verbund* OUTER-JOIN(R, S, P, c) mit $c \in \{\text{left}, \text{right}, \text{full}\}$

- allgemeine Vorgehensweise bei Restrukturierung:
  - komplexe Verbünde, Selektionen in binäre aufteilen
  - Selektion möglichst „weit unten“ ausführen
  - Selektion und Kreuzprodukt zu Verbund gruppieren
  - aufeinander folgende Selektionen der selben Relation zusammenfassen
  - Projektionen möglichst „weit unten“ ausführen (aber Duplikat-Eliminierung vermeiden)
- Planoperatoren (können durch *Pipelining* beschleunigt werden):
  - Selektion (*Scan*):
    * Relationen-Scan: Sequentielles Lesen
    * Index-Scan: Verwendung eines Index
  - Projektion: in andere Planoperatoren integriert
  - Sortierung
  - Join mit Relationen R, S:
    * Nested-Loop-Join (für *Gleichverbund* mit Index-Zugriff verbesserbar)
    * Sorted-Merge-Join (nur für Gleichverbund): sortiere R, S; *schritthaltender* Scan
    * Hash-Join (nur für Gleichverbund): kleinere Relation hashen (bei zu großer Relation mehrere Teile); über größere sequentiellen Scan
  - Duplikat-Eliminierung
  - Gruppierung
- je nach System/Anwendung Optimierung auf niedrige CPU-/IO-Last
- *Statistiken* für Wahl des Planoperators sinnvoll (Verteilung der Tupel, Selektivität, ...)