

Zusammenfassung IDB

Marco Ammon

11. Februar 2019

1 Einführung

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

2 Dateiverwaltung

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

3 Sätze

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

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

4 Schlüssel

- *Schlüsselwerte* als „inhaltsbezogene Adressen“
- *Hashing*:
 - *Hash-Funktion* verteilt Schlüsselwert möglichst gleichmäßig auf verfügbare *Buckets* (Blöcke)
 - *Divisions-Rest-Verfahren*: $h(k) = (k \bmod q)$ mit Schlüsselwert k und Anzahl der Buckets q
 - Problem des *Überlaufs* mit verschiedenen Lösungsmöglichkeiten:
 - * *Open Addressing*: Ausweichen auf Nachbarbuckets
 - * spezielle *Overflow-Buckets*: Bucket verweist auf „*seinen*“ Overflow-Bucket
 - *virtuelles Hashing* zur konstanten Reorganisation:
 - * Anzahl der Buckets q , Sätze pro Bucket $b \Rightarrow$ Kapazität $:= q \cdot b$
 - * Belegungsfaktor $\beta := \frac{\text{Anzahl gespeicherter Sätze } N}{\text{Kapazität}}$
 - * Wenn $\beta >$ Schwellwert 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 die Blattknoten mit direktem Vorgänger S' und direktem Nachfolger S'' von S
- wähle den größeren
- ersetze S je nach Wahl durch S' bzw. S''
- lösche entsprechenden Schlüssel S' bzw. S'' und ggf. Unterlauf behandeln
- *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.executeQuery(ANFRAGE);`
 - außerdem: `int executeUpdate(String sql), boolean execute(String sql)`
 - * `while (ergebnis.next()) {`
 - `int pnr = ergebnis.getInt(1);`
 - `}`
 - Bei mehrfacher Ausführung der gleichen Abfrage mit unterschiedlichen Werten *prepared statements* sinnvoll:
 - * Anfrage enthält Platzhalter für Werte
 - * Analyse, Ausführungsplanerstellung und weiteres wird sofort durchgeführte
 - * JDBC:
 - `PreparedStatement prep = con.prepareStatement(INSERT ...VALUES (?,?))`
 - Setzen der Werte mittels `void setDATENTYP(int paramId, DATENTYP val)`
 - Ausführung mit `prep.executeUpdate()`
 - bei stored-procedures nur noch einmaliges Analysieren, etc. zur Compile-Zeit erforderlich:
 - * Prozedur in DBVS bekommt „Namen“, über den sie mit Werten als Parametern aufrufbar ist

* JDBC:

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

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

7 Transaktionen

- sinnvoll für *nebenläufigen* Zugriff
- erleichtern Umgang mit *Fehlern* und Ausfällen
- *Programmfehler*: Absturz des Datenbank-Anwendungsprogramms \Rightarrow Daten im Puffer und auf Festplatte in undefiniertem Zustand
- *Systemfehler*: DBVS oder BS fällt aus, Hardware-Fehler, ... \Rightarrow Daten im Puffer verloren, auf Festplatte in undefiniertem Zustand
- *Gerätefehler*: Festplattenausfall \Rightarrow Daten auf Festplatte sind verloren
- *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)