Externspeicherverwaltung

- Verwaltung externer Speichermedien
- Verbergen von Geräteeigenschaften
- Abbildung von physischen Blöcken
- Kontrolle des Datentransports von/zum Hauptspeicher
- ggf. Realisierung einer mehrstufigen Speicherhierarchie
- ggf. Fehlertoleranzmaßnahmen (falls kein Standard-RAID verwendet werden soll)

Dateikonzept

Motivation

- selektive Aktivierung von Dateien
- temporäre Dateien
- Einsatz unterschiedlicher Speichermedien
- kurze Adreßlängen

DB-Speicher = Menge von Dateien

Realisierung eines Dateikonzeptes

- Katalog für alle Dateien
- Dateideskriptor: Name, Owner, Zugriffskontrollliste, Größe, Zeitstempel, etc.
- Freispeicherverwaltung für Externspeicher durch formatierte Bitlisten
- Einheit des Zugriffs: Block
  - feste Blocklänge pro Datei
  - Unterstützung für sequentielle E/A Vorgänge

Das alles leistet in der Regel das Betriebssystem.

Blockzuordnungsverfahren

Statische Datei Zuordnung

- direkte Adressierung
- minimale Zugriffskosten
- keine Flexibilität

Dynamische Extent-Zuordnung

- Adressierung über eine kleine Tabelle
- geringe Zugriffskosten
- mäßige Flexibilität (Verschnitt)

Dynamische Block-Zuordnung

- Adressierung über eine große Tabelle
- hohe Zugriffskosten
- maximale Flexibilität

Wie macht es ein Datenbänkler? Wie macht es UNIX?

Effizienz des sequentiellen vs. wahlfreien Zugriffs?

TABLESPACES (Segmente)

- Tablespace wird auf Dateikonzept aufgesetzt
- 1:n Beziehung zu Dateien
- Administrator verwaltet Tablespaces und regelt damit welche Tabellen, wo und wie abgelegt werden
- Anlegen eines Tablespaces

  ```
  CREATE TABLESPACE wutz DATAFILE
  wilde.dat SIZE 4MB, wutz.dat SIZE
  3MB
  ```

- Erweitern eines Tablespaces (erfolgt oft nicht automatisch)

  ```
  ALTER TABLESPACE wutz ADD DATAFILE
  hat.dat SIZE 2MB
  ```

- Zuordnung zu Relationen

  ```
  CREATE TABLE husten(...) TABLESPACE
  wutz
  ```

Seitenzuordnungsverfahren

Direkte Seitenzuordnung

- *update in place*
- direkte Adressierung

Indirekte Seitenzuordnung

- Schreibe (logische) Seite nach Änderung in einen anderen (physischen) Block
- Seitentabelle bildet logische Seite auf physischen Block ab
- mengenorientiertes Einbringen: Umschalten der Seitentabelle

Schattenspeicherkonzept

Ziele

- atomares Einbringen von Änderungen zu einem Sicherungspunkt
- Sicherungspunkte können individuell für Segmente gesetzt werden

Datenstrukturen

- V_{k0}, V_{k1}: 2 Versionen der Seitenzuordnungstabelle für jedes Segment
- Map_0, Map_1: 2 Versionen eines Bitvektors zur Freispeicherverwaltung
- Mapswitch: Bit, das anzeigt, ob Map_0 oder Map_1 aktiv ist
- Status: BitVektor, der anzeigt, welche Segmente geöffnet sind

Algorithmus

Öffnen von Segment k zum Schreiben

1. kopiere V_{k0} nach V_{k1}
2. setze Status[i] = 1
3. schreibe Status und Mapswitch atomar auf Platte
4. lege CMap als Arbeitskopie von aktiver Map an

Erstmalige Änderung von Seite P_i

1. lies Seite P_i (i.e., Block $V_{k0}[i]$)
2. finde freien Block j in CMap
3. setze $V_{k0}[i] = j$
4. markiere Seite P_i in $V_{k0}[i]$ als geändert
5. (jede weitere Änderung direkt durchführbar)

Anlegen einer neuen Seite

- analog zum Ändern

Schließen des Segmentes

- erzeuge Bitliste mit aktueller Speicherbelegung und speichere sie in nicht aktive Map
- schreibe die nicht aktive Map auf Platte
- schreibe V_{k0} auf Platte
- schreibe alle geänderten Blöcke auf Platte
- setze Status[k] = 0; Mapswitch = 1 - Mapswitch
- schreibe Status und Mapswitch atomar auf Platte

Zurücksetzen geöffneter Segmente

1. kopiere V_{k1} nach V_{k0}
2. setze Status[k] = 0

Bewertung des Schattenspeicherkonzeptes

Vorteile

- Rücksetzen sehr einfach und schnell
- erzeugt operationskonsistente Sicherungspunkte
- nach schlimmen Fehler ist die DB mit höherer Wahrscheinlichkeit in einem brauchbaren Zustand als bei einer direkten Einbringstrategie
- ermöglicht logisches Logging für REDO

Nachteile

- Hilfsstrukturen können sehr groß werden ($>$ 1 Block)
- in schlimmen Fällen können sie sogar nicht ganz im Hauptspeicher gehalten werden
- Sicherungspunkt erzwingt Herausschreiben des gesamten DB Puffers
- zusätzlicher Speicherbedarf für Doppelbelegung
- Clusterbildung (sequentieller Zugriff auf verwandte Daten) wird zerstört

Vorsicht: Nicht alle Konzepte, die in dieser Vorlesung vorgestellt werden, taugen in der Praxis etwas!

Pufferverwaltung

Zum warm werden:

In welcher Schicht liegt die Pufferverwaltung?

Menü

- Suche im Puffer; Ablauf des Zugriffs
- satz- vs. seitenorientierte Pufferverwaltung
- Schreib- und Lesestrategien
- Allokation und Ersetzungsverfahren
- DBMS vs. BS

Implementierung eines Zugriffs

Suche

- gewöhnliches Hashing
- (theoretisch auch andere Strategien vorstellbar)

Datenstruktur Handle (ca. 32 Byte)

- *Nummer/Id des Datenobjektes*
- *Modified Bit*
- *Fix Bit*
- *Status für Ersetzungsstrategie*
- *Zeiger auf Datenobjekt im Puffer*
- *Zeiger für Hashtabelle*

Zugriff

- **FIX** des Datenobjektes; das Datenobjekt darf weder ersetzt noch innerhalb des Puffers verschoben werden
- Zugriffe des Anwendungsprogramms oder höherer Schichten des DBMS via Handle
- bei Änderung wird *Modified Bit* gesetzt
- **UNFIX** des Datenobjektes; das Datenobjekt darf wieder zugegriffen werden; *Statusänderung* durch Ersetzungsstrategie

Satz- vs. Seitenorientierte Pufferverwaltung

Seitenorientiert:

- gut: geringer Verwaltungsaufwand (i.e., Hashtabelle und Ersetzungsstrategie)
- gut: einfach zu implementieren
- gut: kein Verschnitt
- gut: kein Kopieraufwand notwendig
- schlecht: Verschwendung durch nutzlose Daten
- schade: nicht immer möglich

Satzorientiert:

- schlecht: Verwaltungsaufwand pro Satz
- schlecht: aufwendige Implementierung mit Buddy System
- schlecht: (interner) Verschnitt beim Buddy System
- schlecht: Kopieraufwand zum Isolieren der Sätze
- schlecht: evtl. Pagefault beim Zurückschreiben eines modifizierten Objektes (installation read)
- gut: puffere nur relevante Sätze
- gut: garantiert immer einsetzbar

Ausblick: duale Pufferverwaltung

Schreibstrategien

- **FORCE.** (alla Write-through)
  - schreibe geänderte Seiten *spätestens* beim *commit* zurück
  - einfache Recovery
  - hoher E/A Aufwand
  - hohe Antwortzeit von Transaktionen

- **NO FORCE.** (alla Write-back)
  - schreibe geänderte Seiten beim *Ersetzen* der Seite zurück
  - viel besser und klappt dank ARIES

Lesestrategien

Preplanning

- analysiere Anwenderprogramme
- materialisiere *Page Answer* in Abhängigkeit von Parametern
- lese *Page Answer* bei Aufruf eines Anwenderprogrammes
- (ich bin kein Fan hiervon)

Prefetching (chained I/O)

- nutze Clusterbildung der Daten aus
- gut: überlappe I/O und CPU Operationen
- schlecht: falsche Entscheidungen
- schlecht: Festplatte macht ohnehin schon *read-ahead*
- schlecht: bringt nichts bei hohen Puffertrefferraten (in diesem Fall nur zusätzlicher Aufwand)

Lesestrategien (ctd)

Demand Fetching

- setze Seitenanforderung ab, wenn Seite wirklich gebraucht wird
- gut: konservativ, kein nutzloser Zusatzaufwand
- schlecht: evtl. hohe Antwortzeiten, da Anwendung blockiert

Referenzmuster

Datenbankanwendung verfolgen typischerweise folgende Referenzmuster:

- **Sequentielle Suche.** Beispiel: table scan; charakerisiert durch Länge l

  $$P_1, P_2, P_3, P_4, \ldots, P_l$$

- **Hierarchische Pfade.** Beispiel: index scan; charakterisierbar durch Markov Ketten (uff!)

  P_1, P_4, P_7 oder P_1, P_5, P_{10}

- **Zufällige Zugriffe.** Beispiel: RID scan; charakterisiert durch Länge l und Seitenmenge $\mathcal{S}$

  $$P_{17}, P_{43}, P_7, P_8, P_{17}, P_{55}, P_3, \ldots$$

- **Zyklische Pfade.** Beispiel: nested-loop join; charakterisiert durch Länge l und Zyklenlänge n

  $$P_1, P_2, \ldots, P_n, P_1, \ldots P_n, \ldots, P_1, \ldots P_n$$

Seitenersetzung

Klassische Seitenersetzungsverfahren

- least-recently used (LRU)
- most-recently used (MRU)
- first-in first our (FIFO)
- clock / second-chance / gclock
- least-frequently used (LFU)
- least-reference density

Wie gut funktionieren diese Verfahren bei den obigen Referenzmustern?

LRU-k

Prinzip

Ersetze Seite, bei dem der k-te letzte Zugriff am längsten zurückliegt.

Bemerkungen

- LRU-1 ist identisch mit klassischem LRU
- Datenobjekte, auf die noch keine k mal zugegriffen wurde, werden nach normalem LRU behandelt
- man muß sich Historie eines Datenobjektes merken, selbst wenn das Datenobjekt bereits ersetzt wurde! (Wie lange ist ein Tuningparameter.)
- Korrelierte Zugriffe dürfen nur einmal registriert werden (z.B. via *Fix-Unfix* Mechanismus) – das ist bei LRU-k von essentieller Bedeutung
- Priority Heap erforderlich
- man kann zeigen, daß LRU-k *optimal* ist (was heißt das eigentlich?)

Frage

Hilft LRU-k in Datenbanksystemen?

2Q

Motivation

- LRU-2 reicht in der Regel
- Annäherung an LRU-2
- weniger Aufwand (siehe Übung)

Verfahren

- A_1 ist FIFO Schlange für alle Seiten die gerade gelesen wurden
- A_m ist LRU Stack für Seiten, die mehr als einmal gelesen wurden
- A_1 und A_m haben eine fixe Größe
- d.h. wenn eine Seite geladen wird, kommt sie in A_1 rein; sie ersetzt dort eine andere Seite
- wenn auf eine Seite aus A_1 zum zweiten Mal zugegriffen wird, kommt sie nach A_m; dort ersetzt sie dann wiederum eine Seite
- (die Ersetzung in A_m kann verzögert stattfinden)

Problem bei 2Q

- gute Größe von A_1 ist schwierig zu bestimmen; ähnlich wie bei LRU-k weiß man nicht, wie lange man Statistiken einer Seite halten muß, bevor man feststellt, daß die Seite *heiß* ist
- leichte Abhilfe: A_1 partitionieren in A_{in} und A_{out}; löst das Problem aber nicht vollständig

Segmentierung des Puffers

Prinzip

- jeder Prozeß (oder Operator) erhält eigenes *logisches* Segment
- verursucht ein Prozeß einen Seitenfehler im DB-Puffer, wird die Seite in das *logische* Segment des Prozesses eingefügt
- (reinlinsen in fremde Prozesse ist allerdings erlaubt)
- jedes Segment hat seine eigene Ersetzungsstrategie
  - *random* für *sequentielle* oder *zufällige* Zugriffsmuster (wieso?)
  - *LRU* für *hierarchische* Zugriffsmuster (wieso?)
  - *MRU* für *zyklische* Zugriffsmuster (wieso?)
- statische Allokation wird für jedes Segment separat durchgeführt;
  - z.B. wird 1 Seitenrahmen für ein sequentielles Zugriffsmuster reserviert
- Lastkontrolle bei Anfragebearbeitung: Optimierer ermittelt Pufferbedarf eines Operators; kann dieser Bedarf gedeckt werden, wird der Operator ausgeführt; ansonsten darf der Operator nicht starten
- Problem: Memory Deadlock bei komplexen Plänen mit Pipelining

Wichtige Bemerkungen

- Segmentierung erlaubt es spezielles Wissen über die Referenzmuster von Datenbanksystemen auszunutzen
- genauer Pufferbedarf eines Prozesses allerdings schwer vorherzusehen weil viele Prozesse auf dieselben Daten zugreifen; dies kann zu einer zu konservativen Lastkontrolle führen (wieso?)

Alternative: Working Set

- Fenster der letzten Zugriffe wird im Puffer gehalten
- Segmente wachsen und schrumpfen dynamisch
- zur Lastkontrolle werden Prozesse blockiert, wenn ihre Segmente wachsen sollen aber kein Pufferplatz zur Verfügung steht
- sehr clever; wird aber in Datenbanksystemen nicht verwendet, weil es eben nicht das spezielle Wissen über die Referenzmuster eines DBS ausnutzt

DB-Puffer und Virtueller Speicher

Beobachtung

- Der DB-Puffer ist auf dem virtuellen Speicher des Betriebssystems implementiert
- Im Grunde führen beide dieselbe Art Aufgabe aus
- Das führt zu unangenehmen Effekten (i.e., zusätzliche I/O beim Swappen des BS).

Double Page Fault

- die benötigte Seite ist nicht im DB-Puffer
- die zu ersetzende Seite (ist im DB-Puffer) ist nicht im BS-Cache
- in diesem Fall muß die zu ersetzende Seite erst vom BS-Swapbereich gelesen werden (das ist natürlich dämlich!)
- Geht es noch schlimmer?