Speicherstrukturen und Zugriffssystem

Speicherhierarchie

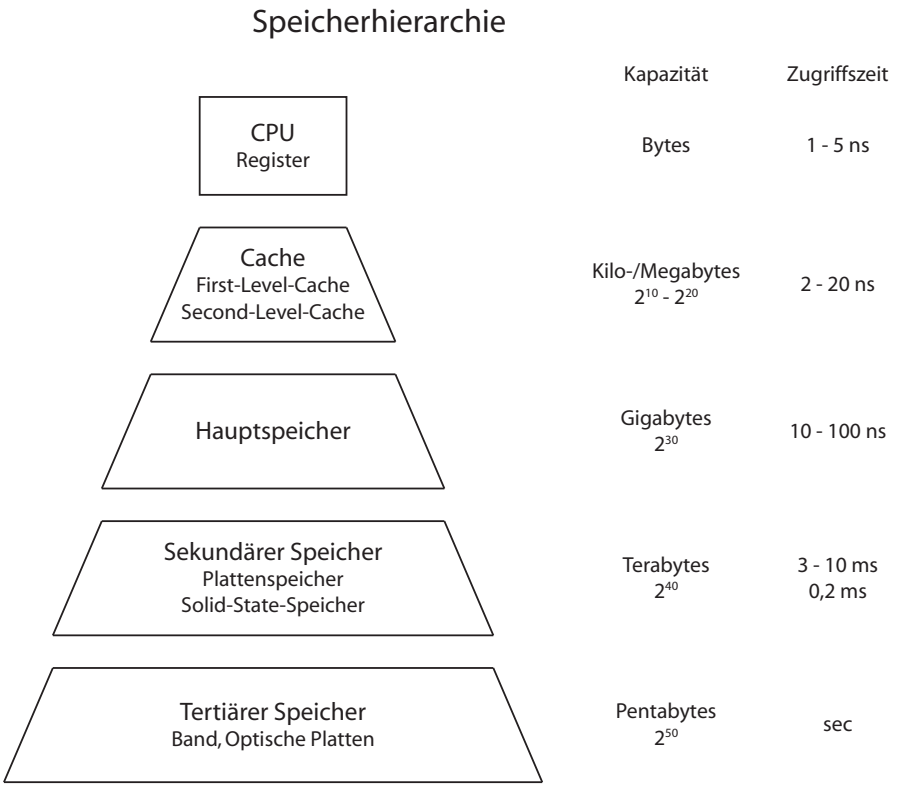

Abbildung 1: Speicherhierarchie

Speicherzugriffe

Ein DBMS transferiert Daten von Festplatten (nicht-volatilem Speicher) in Blöcken (*blocks*), die z.B. 4 KBytes groß sind.

Man geht von folgendem grundlegenden Kostenmodell aus:

Kostenmodell Die Zeit für den Zugriff auf die Festplatte ist um vieles größer als jegliche Datenoperation im Hauptspeicher. D.h. die Zahl der Blöcke, die von der Festplatte in den Hauptspeicher und umgekehrt übertragen werden müssen ist für die Beurteilung der Geschwindigkeit von Algorithmen in DBMS ausschlaggebend.

Dieses Kostenmodell hat Folgen: Stellen wir uns etwa einen Zugriff auf Daten über einen Index vor. Dann müssen die gesuchten Daten zunächst im Index lokalisiert werden und dann erst erfolgt der eigentliche Zugriff auf die gesuchten Daten. Da jeder Zugriff immer komplette Blöcke betrifft, genügt es im Index die Adresse des Blocks zu speichern, detailliertere Information wäre sinnlos.

Präziser ergibt sich die Zugriffszeit auf einen Block auf einer rotierenden Festplatte folgendermaßen:

Zugriffszeit =
  Suchzeit (für Positionierung des Kopfs auf der Spur)
+ Latenzzeit (Rotation zum Sektor mit Blockanfang)
+ Übertragungszeit (Zeit für Lesen/Schreiben des Blocks)

Es ist offensichtlich, dass beim Zugriff auf mehrere Blöcke es am günstigsten ist, wenn sie unmittelbar aufeinander auf der Festplatte angeordnet sind, weil dann die Suchzeit zwischen den einzelnen Blöcken sehr gering ausfällt. Am besten wäre es also, die Daten so zu organisieren, dass sie für jeden Zugriff in sequenziellen Blöcken liegen. Dies geht aber nicht, denn man weiß ja nicht vorher, welche Daten von den Anwendungen eines Datenbanksystems aufeinanderfolgend benötigt werden. Man kann die Daten sortiert nach dem Primärschlüssel auf die Platte schreiben (sequenzielle Datei) oder zusammen mit einem Index, der den Primärschlüssel indiziert (indexsequenzielle Datei). Dadurch wird der Zugriff über Bereiche der Werte des Primärschlüssels beschleunigt. (In Oracle kann man z.B. beim Anlegen einer Tabelle mit der Klausel `organization index` erreichen, dass die Daten nach dem Primärschlüssel sortiert gespeichert werden. Die Dateistruktur, die Oracle dazu verwendet ist jedoch die des B^*-Baums.)

Diskussion In der jüngsten Zeit gibt es zwei Entwicklungen.
(1) Hauptspeicher werden immer größer, so dass es oft möglich ist, die kompletten Daten einer Datenbank in den Hauptspeicher zu laden. (Es gibt heute Server mit 2TB Hauptspeicher.)
(2) Es werden zunehmend Solid-State-Disks eingesetzt. Diese sind sehr viel schneller bei lesenden Zugriffen, etwa gleich schnell (oder schneller) beim Schreiben im Vergleich mit magnetischen Platten.
Für DBMS, die solche Techniken nutzen können, sind möglicherweise etablierte Algorithmen nicht wirklich die geeigneten! Einfluss der technischen Infrastruktur auf Architektur und Implementierung von DBMS!

Storage-Area-Networks Wenn man andererseits sehr viele Daten hat, werden SANs eingesetzt. Es handelt sich um block-basierten Zugriff auf

Speicher über ein Netz. Typischerweise emuliert ein SAN einen Zugriff auf Blöcke einer Platte, ohne dass man über die dahinterliegende Technik (RAID, physische Platten u.ä.) Bescheid wissen muss.

Internet Amazon bietet Zugriff auf beliebig viel Speicher über den *Amazon simple storage service* S3. Man greift auf S3 mittels Webservices zu. Einige Versuche S3 als „Speichermedium“ für ein DBMS zu nutzen findet man in Matthias Brantner, Daniel Florescu, David Graf, Donald Kossmann, Tim Kraska: Building a Database on S3 in: *Proceedings of the 2008 ACM SIGMOD international conference on Management of Data.*

Datensatzformate

Wenn man die Organisation der Daten auf dem Sekundärspeicher untersucht, spricht man von *Dateien* und *Blöcken*. Hat man die Organisation der Daten in der Pufferverwaltung im Auge spricht man von *Segmenten* und *Seiten*. Aus der Sicht des Zugriffssystems stellt man sich eine Relation bzw. Tabelle als eine *Datei* vor, die Tupel bzw. Zeilen enthält, die man als *Datensätze* bezeichnet. Jeder Datensatz enthält die *Werte*, die zu den Attributen der Relation bzw. Feldern der Tabelle gehören.

Wir betrachten im Folgenden drei Formen von *Datensatzformaten*: Datensätze fixer Länge, Datensätze variabler Länge und Datensätze mit sehr großen Datenfeldern.

Dabei betrachten wir die Organisation der Datensätze in Seiten, und unterstellen dabei, dass diese im Prinzip der Organisation in Blöcken entspricht.

Diskussion Diese Annahme ist bei vielen gängigen Implementierungen von Datenbankmanagementsystemen durchaus zutreffend. Sie ist aber keineswegs zwingend. C.J. Date berichtet in der 8. Ausgabe seines Buchs „An Introduction to Database Systems“ über das sogenannte „TransRelational Model“, das eine ganz andere physische Organisation der Datensätze einer Relation vorsieht. MonetDB (`http://www.monetdb.org/Home`) ist ein relationales Datenbankmanagementsystem, das das sogenannte *Decomposition Storage Model (DSM)* verwendet.[1] Wir werden im letzten Teil der Vorlesung kurz auf diese Techniken eingehen.

[1] In diesem Modell wird eine n-stellige Relation in binäre Relationen aufgelöst und als solche gespeichert, es handelt sich also um eine *spalten-orientierte* Organisation. Siehe George P. Copeland, Setrag N. Khoshafian: *A Decomposition Storage Model*, Proceedings of the 1985 ACM SIGMOD international conference on Management of data, p.268-279, May 1985, Austin, Texas, USA sowie Peter A. Boncz, Martin L. Kersten, Stefan Manegold: *Breaking the memory wall in MonetDB*, Communications of the ACM, Volume 51 Issue 12, December 2008.

Datensätze fixer Länge

In Datensätzen fester Länge haben alle Datenfelder ein fix definierte Länge. Eine Seite ist dann typischerweise so organisiert:

Jede Seite enthält dieselbe Zahl von Datensätzen. Jeder dieser Datensätze hat denselben Aufbau, die Offsets innerhalb eines Datensatzes können aus der Kataloginformation über den Aufbau des Datensatzes ermittelt werden.

Jeder Datensatz kann identifiziert werden durch die Nummer der Seite sowie die Position des Datensatzes innerhalb der Seite.

Typischerweise enthält eine Seite ein Bitfeld, in dem verzeichnet ist, welche der Datensätze in der Seite gültig sind und welche gelöscht oder noch nie benutzt sind.

Datensätze variabler Länge

In Datensätzen variabler Länge können die Datenfelder unterschiedliche Länge haben.

Die Felder in den Datensätzen werden so angeordnet, dass zunächst die Felder fixer Länge kommen und dann diejenigen variabler Länge kommen. Zu Beginn jedes Datensätzes folgen nach Kopfinformationen (etwa Datensatzlänge) die Offsets der Felder variabler Länge.

Jede Seite enthält ein Verzeichnis der Datensätze sowie die Anzahl der Datensätze. Das Verzeichnis enthält den Offset des Datensatzes innerhalb der Seite.

Datensätze, die Seiten überspannen

Es kann vorkommen, dass die Datensatzlänge bei gegebener Seitengröße zu extrem viel Verschnitt an Speicherplatz führt. Oder in der Datenbank werden Felder gespeichert, deren Größe die Seitengröße übersteigt, etwa BLOBs. In diesem Fall werden in den Seiten Fragmente gespeichert mit doppelt verketteten Referenzen auf die Folge der Fragmente.

Schichten des Datenzugriffs

Übersicht

Der Zugriff auf persistente Daten in einem DBMS ist in drei Schichten organisiert:

Das *Zugriffssystem* sieht die Daten in der Datenbank als Dateien (*files*),

die Datensätze (*records*) enthalten. Eine Datei von Datensätzen entspricht einer Tabelle mit ihren Zeilen. (Hier ist von Dateien die Rede in einem allgemeineren Sinne als von Dateien im Betriebssystem. Manche DBMS verwenden dazu tatsächlich Dateien des Betriebssystems mit den dadurch gegebenen Funktionen, andere DBMS implementieren Dateien selbst. Wir verstehen hier Dateien als konzeptuellen Begriff.) Darüberhinaus kennt das Zugriffssystem Indexe, die wir später betrachten.

Im Prinzip kann man sich vorstellen, dass die Datensätze einer Datei in Seiten organisiert sind. Die *Pufferverwaltung* hält diese Seiten im Speicher. Sie vermittelt zwischen der Sicht auf die Puffer als Bestandteile von Dateien und der tatsächlichen Speicherung der Seiten als Blöcke auf dem Plattenspeicher.

Die *Speicherverwaltung* ist für den eigentlichen Transfer der Daten von und zur Platte zuständig.

Im Folgenden betrachten wir diese drei Schichten „von unten nach oben“:

Speicherverwaltung

Die Speicherverwaltung ist für den eigentlichen Zugriff auf die Daten auf externen Speichermedien (bei uns in der Regel Festplatten) zuständig.

Sie verbirgt gegenüber den höheren Schichten die Details des I/O-Zugriffs und stellt ihnen die Daten als eine Sammlung von Blöcken dar. Der Block mit typischerweise 2-32 KBytes Größe ist die Sichtweise, die die Speicherverwaltung anbietet.

Die Speicherverwaltung stellt also folgende Routinen bereit:

- `allocate/deallocate` zur Bereitstellung von Speicher für Blöcke.
- `read/write` zum Lesen und Schreiben von Blöcken.

Die Speicherverwaltung muss also eine Zuordnung von Blöcken wie sie in der Pufferverwaltung angesprochen werden zur eigentlichen physischen Speicherung vornehmen.

Die physische Adresse besteht z.B. aus

- Identität der Platte, auf der der Block gespeichert ist
- Zylinder, Track und Blocknummer

(Werden innerhalb des DBMS Pointer verwendet wie in OODBMS oder sog. Referenzen wie in ORDBMS, muss es eine Zuordnung zwischen den Adressen im Speicher und den Adressen in der Datenbank geben. Hierfür

gibt es verschiedene Mechanismen der Transformation von Referenzen in Hauptspeicheradressen, die unter dem Terminus *Pointer-Swizzling* beschrieben werden.)

Pufferverwaltung

Die *Pufferverwaltung* kümmert sich um den Datenbankpuffer (*buffer pool*, auch Cache genannt) im Hauptspeicher.

Der Datenbankpuffer besteht aus sog. Seitenrahmen (*memory frames*), die eine Seite enthalten. Die Blöcke werden von der Platte in solche Rahmen geschrieben und stehen dort für lesende und schreibende Zugriffe zur Verfügung. Ein *Ersetzungsverfahren* kümmert sich darum, welche Seiten im Puffer ersetzt werden können, wenn kein Platz mehr für weitere angeforderte Seiten vorhanden ist. Dadurch kümmert sich die Pufferverwaltung gleichzeitig um die Synchronisierung zwischen Puffer und persistentem Speicher.

Verwender der Pufferverwaltung müssen mitteilen, wie sie Blöcke verwenden möchten. Dazu hat die Pufferverwaltung die Funktionen `pin` bzw. `unpin` zum Fixieren von Seiten bzw. Freigeben einer Fixierung. Dies ist notwendig, weil Verwender der Pufferverwaltung die Daten im Puffer direkt ändern können, also Pointer auf Adressen in diesen Seite halten.

Ein Beispiel einer lesenden Transaktion:

```
addr = pin(p);
...
... lesende Zugriffe aüf die Seite an addr
...
unpin(p, false);
```

Die Schnittstelle zur Pufferverwaltung besteht also aus

- `pin(pageno)`
  fordert die Seite mit Nummer `pageno` an, er wird ggfs. in den Puffer geladen, die Seite wird als `clean` markiert (bedeutet identisch mit persistentem Inhalt) und es wird die Adresse des Frames zurückgegeben.
- `unpin(pageno, dirty)`
  gibt die Seite an die Pufferverwaltung zurück, dabei wird das Dirty-Bit gesetzt, falls die Seite modifiziert wurde. Dadurch weiß die Pufferverwaltung, ob sie beim Ersetzen der Seite die Daten in der Speicherverwaltung schreiben muss.

Pseudocode für die Implementierung von `pin` und `unpin`:

```
address pin( pageno ) {
  if (Puffer enthält Seite mit pageno) {
    pinCount(pageno)++;
    return Adresse von Frame mit dem Block
  } else {
    Suche ein Frame f, der ersetzt werden kann
    if ( dirty(pageno in f) ) {
      Schreibe Seite in f auf Platte;
    }
    Lese Seite mit pageno von Platte in Frame f;
    pinCount(pageno) = 1;
    dirty(pageno) = false;
    return Adresse von Frame f
}

void unpin(Pageno, dirty) {
  pinCount(pageno)--;
  dirty(pageno) = dirty(pageno) || dirty;
}
```

Welche Probleme können auftreten?

Bei Änderung von Dateien z.B. beim Insert kann es vorkommen, dass eine Seite nicht mehr ausreicht, um die neuen Datensätze aufzunehmen. Dann muss eine neue Seite angelegt werden...

Angenommen mehrere Transaktionen verwenden dieselbe Seite und versuchen dieselben Daten zu verändern. Dann tritt möglicherweise ein Konflikt beim Zugriff auf. Dieser Konflikt muss durch die Synchronisationskontrolle reguliert werden, *ehe* die Funktion `pin` für die Seite verwendet wird.

Techniken der Pufferverwaltung:

- Prefetching
  Seiten, die mit großer Wahrscheinlichkeit verwendet werden, werden vorsorglich in den Puffer geladen.
  Z.B. vermutet man sequenzielles Lesen und liest automatisch immer einige Blöcke voraus oder ein Datenbankalgorithmus gibt der Pufferverwaltung Informationen über die benötigten Blöcke.

- Partitionierung
  Der Puffer wird unterteilt und es werden getrennte Verwaltungen für z.B. Seiten von Daten aus Tabellen und Daten aus Indexen verwendet.

Ersetzungstechniken:

- LRU (*Least Recently Used*)
  ersetzt diejenige Seite im Puffer mit pinCount == 0, die am längsten nicht mehr angesprochen wurde. Dazu benötigt die Pufferverwaltung eine Liste der Zeitpunkte der Zugriffe auf die Seiten, was einen relativ hohen Verwaltungsaufwand bedeutet.

- FIFO (*First-In-First-Out*)
  ersetzt die älteste Seite. Diese Strategie benötigt wenig Verwaltung, kann aber zu Merkwürdigkeiten führen. Wir werden den B^+-Baum behandeln, bei dem die Wurzel stets der erste Block ist, der aber eigentlich immerzu benötigt wird. In einer solchen Situation kann FIFO zu unnötigem Ersetzen führen.

- Uhr-Algorithmus („2. Chance“)
  Man stelle sich die Seiten im Puffer im Kreis angeordnet vor und ein Zeiger zeigt auf eine, die aktuelle Seite. Jede Seite hat ein Flag: 0 oder 1. Wenn eine Seite eingelesen wird, erhält sie eine 1, wenn auf sie zugegriffen wird, erhält sich auch die 1. D.h. 1 bedeutet: wird wahrscheinlich noch gebraucht.
  Muss nun eine Seite ersetzt werden, wird der Zeiger verwendet. Er wird bewegt und dabei wird das Flag jeder Seite mit 1 auf 0 gesetzt, dies geschieht so lange, bis eine Seite mit Flag 0 gefunden wird, die dann ersetzt wird.

Zugriffssystem

Die einfachste, aber auch wichtigste Art einer Datei des Zugriffssystems ist die Haufendatei (*Heap File*). In einer Haufendatei werden die Datensätze einfach fortfolgend (ohne Sortierung) gespeichert. Jede Haufendatei wird unter einer Id angesprochen und jeder Datensatz in einer Haufendatei durch eine Satz-Id (*Record id* rid).

Typische Funktionen mit einer Haufendatei:

- erzeuge bzw lösche Haufendatei mit Identität id
  `createFile(id)`, `deleteFile(id)`
- füge eine Datensatz r ein und gebe seine Satz-Id zurück
  `rid insertRecord(f, r)`
- lösche den Datensatz mit der Satz-Id rid
  `deleteRecord(f, rid)`
- liefere den Datensatz mit der Satz-ID rid
  `r getRecord(f, rid)`
- ändere den Datensatz mit der Satz-Id rid
  `updateRecord(f, rid, new_r)`

- initialisiere sequenzielles Durchlaufen
  `initScan(f)`
- gehe zum nächsten Datensatz beim Durchlaufen
  `boolean nextRecord(f)`
- liefere aktuellen Datensatz beim Durchlaufen
  `r getCurrent(f)`

Eine weitere Organisationsform einer Datei ist die *sortierte Datei*, die aufgebaut ist wie eine Haufendatei, in der die Datensätze jedoch sortiert abgelegt sind, etwa nach dem Primärschlüssel. Genauer unterscheidet man *sequenzielle* Dateien und *indexsequenzielle Daten*, siehe [1, S.148ff]

Eine dritte Form der Datei ist die *Hash-Datei*. Gegeben sei eine Hashfunktion $h(x)$, die jeden Datensatz x auf ein ganze Zahl im Intervall $[0, n-1]$ für ein n abbildet. Dabei kann die Hashfunktion als Argument den gesamten Datensatz verwenden, vielleicht aber auch nur den Wert eines Primärschlüssels. Die Zahl $h(x)$ für einen Datensatz x nennt man die *Behälternummer*.

Beispiel für eine Hash-Funktion h: Man nimmt die ersten n Bits des ersten Felds eines Datensatzes. Ist etwa $n = 3$ und das erste Feld ein `int`, dann ist etwa $h(42) = 2, h(14) = 6 \ldots$.

Die Hashfunktion bestimmt die Nummer der Seite, in die ein Datensatz gehört, die Anordnung der Datensätze innerhalb einer Seite ist beliebig.

Ist eine Seite mit Seitennummer p voll, es gibt jedoch weitere Datensätze mit der Behälternummer p, wird eine Kette von Überlauf-Seiten angelegt. Jede dieser Ketten ist also ein *Behälter* (*bucket*) in Bezug auf die Hashfunktion.

Um beim Einfügen in die Seiten etwas „Luft“ zu haben, werden Seiten in Hashdateien beim Anlegen oft nicht gleich voll gefüllt, sondern nur bis zu einem bestimmten Anteil, z.B. 80%. (siehe auch [1, S.182ff])

Literaturverzeichnis

[1] Gunter Saake, Andreas Heuer, Kai-Uwe Sattler. *Datenbanken: Implementierungstechniken.* Heidelberg, 2011.

Burkhardt Renz
TH Mittelhessen
Fachbereich MNI
Wiesenstr. 14
D-35390 Gießen

Rev 3.5 – 14. April 2014