Prof. Dr. Stefan Brass
Institut für Informatik
MLU Halle-Wittenberg

20. November 2015

# Datenbanken II B: DBMS-Implementierung
## — Hausaufgabe 5 —

a) Beschaffen Sie sich den Artikel "Principles of Database Buffer Management" von Wolfgang Effelsberg und Theo Haerder, erschienen in den ACM Transactions of Database Systems, Vol. 9, No. 4, Dezember 1984, Seiten 560–595. Von Rechnern der Universität aus können Sie die ACM Digital Library kostenlos zugreifen, diesen Artikel bekommen Sie über die URI/DOI:

[`http://dx.doi.org/10.1145/1994.2022`]

Verschaffen Sie sich einen kurzen Überblick über den Inhalt. Nennen Sie eine Puffer-Verdrängungsstrategie außer LRU, die in dem Artikel behandelt wird, und beschreiben Sie diese mit wenigen Sätzen.

b) Definieren Sie dann eine Klasse `buf_c` in C++ mit folgenden Komponenten:

- Eine Konstante `BUF_MAXBUFS` für die maximale Anzahl Blöcke, die gleichzeitig im Hauptspeicher gehalten werden können (also die Anzahl Pufferrahmen, d.h. die Cache-Größe).

  Jedes Objekt der Klasse soll einen Pufferrahmen repräsentieren. Eigentlich müsste jedes Objekt also Speicherplatz für einen Block enthalten (oder einen Zeiger auf den Pufferrahmen enthalten). Zur Vereinfachung laden wir die Blöcke aber nicht tatsächlich und simulieren den Puffer nur. Es kann maximal `BUF_MAXBUFS` Objekte dieser Klasse geben. Sie können entweder nach Bedarf Objekte anlegen bis zu dieser Maximalzahl, oder gleich `BUF_MAXBUFS` Objekte erzeugen (im einfachsten Fall mit einem Array).

- Eine statische Methode (Klassenmethode) `init()`, die Sie für Initialisierungen nutzen können. Diese Methode wird vom Hauptprogramm vor allen anderen Methoden aufgerufen. Es ist auch garantiert, dass sie nur ein Mal aufgerufen wird.

- Eine statische Methode (Klassenmethode) `pin`, die zu gegebener Datei-ID (kleine nicht-negative ganze Zahl, Typ `int`) und Blocknummer (nicht-negative ganze Zahl, Typ `int`) einen Zeiger auf ein `buf_c`-Objekt liefert, das den betreffenden Block enthält. Falls der gewüschte Block schon (virtuell) in einem Pufferrahmen steht, soll natürlich dieses Objekt geliefert werden. Ansonsten muss ein neuer Pufferrahmen für den Block belegt werden. Falls es keinen freien Puffer gibt, und wegen der Grenze `BUF_MAXBUFS` kein weiterer Hauptspeicher mehr angefordert werden kann, soll nach der LRU Verdrängungsstrategie versucht werden, einen Puffer frei zu machen. Falls das nicht möglich ist (zu viele `pin` ohne `unpin`), soll

der Nullzeiger zurückgeliefert werden. Bei Mehrbenutzerbetrieb (den wir noch nicht haben) wäre auch denkbar, einige Zeit zu warten und es dann erneut zu probieren.

- Methoden `file_id()` und `block_no()`, mit denen man den im Objekt gespeicherten Block abfragen kann. Solange der Block im Puffer gepinnt ist, dürfen sich diese Werte natürlich nicht ändern. Wenn im Puffer kein Block mit `pin` festgelegt ist, könnte man jeglichen Zugriff als Fehler behandeln (keine Bedingung).
- Eine Methode `unpin()`, die den Block im Puffer freigibt. Beachten Sie, dass auch mehrere Pins für den gleichen Block vorkommen können. Der Puffer ist dann erst wieder frei, wenn es gleich viele Aufrufe von `unpin()` gegeben hat.
- Statische Methoden `hits()` und `misses()`, die die entsprechenden Anzahlen ausgeben.
- Eine statische Methode `clear()`, die alle Puffer wieder als unbenutzt kennzeichnet. Das ist mindestens für Tests nützlich, wenn Sie verschiedene Tests hintereinander durchführen wollen. Nach Aufruf von `clear()` sollen auch `hits()` und `misses()` wieder `0` liefern.

In einem realen Puffer-Manager brauchte man mindestens noch folgende Methoden (nicht Bestandteil der Hausaufgabe):

- Eine Methode `contents()`, die einen Zeiger auf den Datenblock im Puffer liefert (für Blöcke hätte man natürlich auch eine Oberklasse `block_c` und Unterklassen je nach Inhalt, z.B. für Blöcke einer Heap-Datei oder eines B-Baums).
- Eine Methode `set_modified()`, die den Inhalt des Puffers als verändert markiert. Der Puffer kann dann nur wiederverwendet werden, nachdem der Block in die Datei zurückgeschrieben wurde.
- Unterstützung für das Setzen eines Checkpoints, bei dem alle veränderten Blöcke in die Datei zurückgeschrieben werden. Richtig gut geht das wohl nur mit mehreren Threads, so dass man neben der eigentlichen Bearbeitung von Anfragen und Updates gelegentlich Blöcke zurückschreibt. Minimallösung wäre eine statische Methode `checkpoint()`, die alle veränderten Puffer in die jeweilige Datei zurückschreibt.
- Außerdem braucht man Unterstützung für temporäre Daten. Diese müssen nicht unbedingt in eine Datei geschrieben werden, solange sie aus dem Puffer nicht verdrängt werden. Eine Minimallösung wäre, temporäre Daten nie aus dem Puffer zu verdrängen. Das setzt aber Grenzen z.B. für Sortieraufgaben. Besser wäre es, wenn eine oder mehrere Dateien für die eventuell notwendige Auslagerung benutzt werden könnten.

Testen Sie Ihre Pufferverwaltung mit den Zugriffen aus

[`http://www.informatik.uni-halle.de/~brass/dbi15/refstring.txt`]

und einer Puffergröße (`BUF_MAXBUFS`) von 2000 Blöcken. Wie groß ist die Hit Ratio? Sie können sich ein Rahmenprogramm, das die Testdatei einliest und entsprechend die Methoden `pin` und `unpin` aufruft, unter folgender Adresse herunterladen:

[`http://www.informatik.uni-halle.de/~brass/dbi15/h5_buf/`]

Das Rahmenprogramm ist `buftest.cpp`. Das Verzeichnis enthält noch weitere Dateien, die möglicherweise nützlich sind, u.a. ein `Makefile` und eine erste Version von `buf.h` und `buf.cpp`, die aber im wesentlichen nichts tut. So können Sie das Programm aber schon compilieren.

Es empfiehlt sich natürlich, weitere Tests zu programmieren (eventuell können wir diese später tauschen). Sie können dazu gerne `buftest.cpp` erweitern.

Beachten Sie, dass Ihr Puffermanager natürlich effizient arbeiten soll. Da `pin` eine häufige Operation ist, sollte man versuchen, einen eventuell schon im Puffer vorhandenen Block schnell zu finden. (Dies ist formal keine Bedingung zu Anerkennung der Hausaufgabe, zumindest in gewissen Grenzen.)