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

30. April 2013

Datenbanken II B: DBMS-Implementierung — Hausaufgabe 3 —

Diese Woche soll das Lesen und Schreiben von Blöcken aus einer Datei implementiert werden. Dazu muss aber auch geklärt werden, wie mit eventuellen Fehlern umgegangen werden soll. Wenn der Benutzer eine Operation auf relativ hoher Abstraktionsebene anstößt, wie etwa das Lesen des nächsten Tupels in einem Scan, könnte das dazu führen, dass ein neuer Block gelesen werden muß, und diese Operation könnte fehlschlagen oder feststellen, dass der Block zerstört ist. Der Benutzer muss nun erfahren, dass die Operation gescheitert ist, und sollte aber auch die Möglichkeit haben, eine genauere Fehlermeldung zu bekommen, die z.B. die Nummer des beschädigten Blockes enthält. In diesem Zusammenhang ist auch zu überlegen, ob die Texte von Fehlermeldungen über das ganze Programm verstreut werden sollen, oder an zentraler Stelle gesammelt werden (das würde eine Übersetzung vereinfachen). Mein Vorschlag zum Umgang mit Fehlern ist folgender:

- Jede Methode zeigt über ihren Rückgabewert an, ob sie erfolgreich ausgeführt wurde. Wenn sie keinen anderen Rückgabewert hat, würde man den Typ `bool` wählen, und den Wert `true` für erfolgreiche Ausführung und `false` für Fehler. Wenn ein Zeiger zurückgegeben werden soll, würde ein Null-Zeiger einen Fehler bedeuten. Bei einem Zahlwert würde `-1` einen Fehler anzeigen.

- Außerdem gibt es eine Klasse `err_c` mit nur statischen Methoden (d.h. ohne Instanzen), die die letzten 10 Fehlermeldungen speichert.

  Es gibt eine Methode `err_c::reset()`, die den Speicher von Fehlermeldungen löscht, und eine Methode `err_c::ok()`, die `true` liefert, wenn der Speicher der Fehlermeldungen leer ist. Mit `err_c::num_errs()` kann man die Anzahl der gespeicherten Fehlermeldungen abfragen, und mit `err_c::msg(i)` den Text der `i`-te Fehlermeldung holen. Da die erste Fehlermeldung (mit Index `0`) besonders aufschlussreich ist, sollte wohl auch `err_c::msg()` möglich sein, und diese Meldung liefern.

- In der Klasse `err_c` könnte es auch eine Methode `print` geben, die die Fehlermeldungen nach `cerr` ausdruckt. Außerdem könnte man mit `open_alert(filename)` eine Alert-Datei eröffnen, in die anfallende Fehlermeldungen geschrieben werden (eventuell mit Zeitstempel). Mit `close_alert()` kann man dieses Protokoll der Fehlermeldungen wieder abschalten. Wenn die Datei beim Eröffnen bereits existiert, sollten neue Fehlermeldungen hinten angehängt werden.

- Außerdem enthält die Klasse `err_c` eine große Anzahl von Methoden für die Fehlermeldungen, im Prinzip eine Methode für jeden Typ von Fehlermeldung. Diese Methoden haben Parameter für Daten, die in der Fehlermeldung ausgegeben werden sollen. Z.B. wird `err_c::open_failed(filename)` aufgerufen, wenn die angegebene Datei nicht eröffnet werden kann. Diese Methode könnte dann auch die Betriebssystem-Fehlermeldung abfragen, die möglicherweise zusätzliche Informationen enthält. Alle Methoden für Fehlermeldungen haben den Rückgabetyp `void`.

Das ist nur ein Vorschlag. Sie haben selbstverständlich auch die Möglichkeit, mit Exceptions zu arbeiten. Dann muß man nicht immer den Rückgabewert der Methoden-Aufrufe abfragen. Auf der anderen Seite ist ein Risiko, dass man bei der Programmierung vergisst, dass aus einem Methodenaufruf unerwartet herausgespungen werden kann, und man so vielleicht eine Datenstruktur in einem inkonsistenten Zustand zurückläßt (z.B. nur partiell initialisiert). Wenn man ein Rollback ausführen könnte (das auch Datenstrukturen im Hauptspeicher zurücksetzt), wäre das vermutlich kein Problem. Aber diese Möglichkeit ist im Moment nicht vorgesehen. Es ist aber Ihre Entscheidung.

a) Entscheiden Sie sich für eine Methode zum Umgang mit Fehlern und implementieren Sie die notwendigen Grundlagen (zusätzliche Fehlermeldungen werden natürlich im Laufe des Projektes hinzu kommen). Wenn Sie wollen, können Sie die Alert-Datei weglassen.

b) Legen Sie eine Subklasse von `block_c` für den ersten Block der Datei an ("File Header Block"). Diese Subklasse könnte z.B. `fhblk_c` heissen und in Dateien `fhblk.h` und `fhblk.cpp` definiert sein. Im Laufe der Zeit wird diese Klasse noch wachsen müssen, im Moment soll sie nur die Länge der Datei (in Blöcken) zusätzlich zu den Daten eines normalen Blockes enthalten.

c) Implementieren Sie bitte eine Klasse `file_c` (in Dateien `file.h` und `file.cpp`) mit folgenden Methoden:

   - Einen Konstruktor mit dem Dateinamen als Argument.
   - Eine Methode `filename()`, die den Dateinamen liefert.
   - Eine Methode `create(int blocks)` mit einer Anzahl Blöcken als Argument. Diese Methode soll die Datei mit `blocks * VER_BLOCKSIZE` Bytes beschreiben. Die Blöcke sollen korrekt initialisiert sein, der erste Block (mit der Nummer `0`) also ein "File Header Block" sein, die übrigen Blöcke vom Typ "`BTYPE_EMPTY`". Alle Methoden sollen `true` zurückliefern, falls der Aufruf erfolgreich war, und `false` im Fehlerfall. Diese Methode darf nur aufgerufen werden, wenn die Datei nicht bereits geöffnet ist. Bei solchen Programmierfehlern dürfen Sie das Programm abbrechen (z.B. durch Verwendung von "`assert.h`" oder "`check.h`").
   - Eine Methode `open`, die die Datei zum Lesen und Schreiben eröffnet. Sie muß vorher geschlossen sein.
   - Eine Methode `close` zum Schließen. Die Datei muß vorher geöffnet gewesen sein.
   - Eine Methode `read` mit den Parametern Blocknummer "`int blockno`" und Hauptspeicheradresse (ein Zeiger `buf` auf ein Objekt vom Typ `block_c`): Der betreffende Block soll von der Datei in den Hauptspeicher ab Adresse `buf`gelesen werden. Die Datei muß natürlich geöffnet sein.

- Eine Methode `write(blockno, buf)` mit den gleichen Parametern wie `read`: Der betreffende Block soll vom Hauptspeicher (ab Adresse `buf`) in die Datei geschrieben werden. Die Datei muß natürlich geöffnet sein.
- Eine Methode `sync` ohne Parameter, die sicherstellt, dass alle mit `write` geschriebenen Blöcke auch wirklich auf die Platte geschrieben wurden, und nicht etwa noch in Puffern des Betriebssystems auf das eigentliche Schreiben warten. Die Datei muß geöffnet sein.
- Eine Methode `extend` mit einem Parameter `blocks`, der die Datei um diese Anzahl Blöcke verlängert. Die Datei muß dazu vorher geöffnet worden sein. Außerdem muss der File Header Block angepasst werden, darin steht ja die Größe der Datei.
- Weil man den File Header Block eventuell häufig braucht (sofern er auch auch Informationen zur Freispeicherverwaltung enthält), schien es mir richtig, in den `file_c`-Objekten auch den File Header Block zu puffern. Damit steht er immer zur Verfügung, solange die Datei eröffnet ist. Man könnte ihn auch im normalen Puffer für Datenbank-Blöcke halten, dann würden sich aber zyklische Abhängigkeiten zwischen den Modulen ergeben (die natürlich immer möglich sind). Ich habe in der Klasse `file_c` eine Methode `header()`, die den File Header Block liefert. Ausserdem eine Methode `set_modified()`, mit der man markieren kann, dass der File Header Block verändert wurde, man ihn also später zurück in die Datei schreiben muss. Schließlich noch eine Methode `write_header()`, die das erledigt. Sie würde auch beim Schließen der Datei automatisch aufgerufen, falls es änderungen seit dem letzten Schreiben gegeben hat.
- Natürlich würde im Debug-Modus auch wieder eine Methode `valid()` existieren, die die Konsistenz der `file_c`-Objekte prüft. Im Debug-Modus speichere ich auch an den Anfang jeden Objektes einen `long`-Wert, der die Klasse identifiziert ("Magic Number"). Im Destruktor würde diese Wert ungültig gemacht. Auf diese Art besteht die Möglichkeit, bereits freigegebene Objekte oder manche anderen Pointer-Fehler zu erkennen, die in C++ sonst eventuell länger gesucht werden müssen.

Auch dies ist aber nur ein Vorschlag. Die notwendige Grundfunktionalität ist das Anlegen von DB-Dateien, die Eröffnung, sowie das Lesen und Schreiben von Blöcken. Es sollte möglich sein, im Programm auch mit mehreren DB-Dateien arbeiten zu können, wenn auch jede Datei zunächst eine eigene Datenbank ist.

Beachten Sie bitte, dass, wenn Sie den Betriebssystem-Aufruf `read` in einer Klasse verwenden wollen, die selbst eine Methode `read` hat, Sie `::read` schreiben müssen.