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

22. April 2013

Datenbanken II B: DBMS-Implementierung — Hausaufgabe 2 —

Wie angekündigt, soll in dieser Vorlesung ein Mini-DBMS selbst programmiert werden. Es wird keine direkte Benutzer-Schnittstelle haben, sondern eine Programmier-Schnittstelle (API). Unser "DBMS" könnte später also in eigenen Programmen als Unterprogramm-Bibliothek eingesetzt werden. Es erlaubt, mehrere Relationen in einer von unserer Bibliothek erstellten Datenbank-Datei zu speichern, und per "Full Table Scan" zu lesen. Index-Strukturen sind im Moment nicht vorgesehen, aber späteren Erweiterungen sind natürlich keine Grenzen gesetzt. Physische Zeiger auf einzelne Zeilen werden voraussichtlich implementiert, damit werden auch etwas interessantere Datenstrukturen auf Nutzerebene möglich. Voraussichtlich werden nur Strings fester Länge (also der Typ `CHAR(`n`)`) unterstützt, nicht Strings variabler Länge. Dadurch hätten die Tabellenzeilen auch eine feste Länge. Eine weitere drastische Vereinfachung ist, dass parallele Zugriffe ausgeschlossen werden, ebenso wie eine Wiederherstellung nach einem Systemabsturz. Der Schutz der Daten in der Datenbank ist ja eine wesentliche Funktion eines DBMS, diesen Problembereich würden wir zunächst ausklammern. Es lohnt sich aber, über mögliche Erweiterungen nachzudenken. Man kann auch viel dabei lernen, wenn man ein DBMS plant. Die Zeit reicht nicht, alles im Programm umzusetzen, aber man sollte sich natürlich auch mit anderen DBMS-Funktionen auseinandersetzen.

Ich werde recht detaillierte Vorschläge für Schnittstellen machen, aber es steht Ihnen frei, davon abzuweichen, sofern Sie die wesentliche Funktionalität implementieren. Sie sollten eventuelle Abweichungen allerdings nennen und ausreichend dokumentieren. Die Übungen sind auch eine gute Gelegenheit, Alternativen zu diskutieren. Ich will Ihnen nicht meinen Programmierstil aufzwingen. Manchmal gibt es auch keine perfekte Lösung, so dass man bei Abwägung der Vor- und Nachteile zu unterschiedlichen Ergebnissen kommen kann.

Wenn Sie sich an die vorgeschlagene Schnittstelle halten, könnten Sie ggf. auch meine Lösung (oder eine von Ihren Mitstudenten) verwenden, falls Sie mal eine Hausaufgabe zeitlich nicht schaffen. Ansonsten werde ich eine eindeutig eigene Lösung zu den Programmieraufgaben auch mit maximal einer Woche Verspätung akzeptieren, aber Sie sollten darauf achten, dass Sie nicht abgehängt werden.

Zu meinen C++-Programmierstil gehört Folgendes:

- Ich habe eine Include-Datei `str.h`, in der ein Typ `str_t` für C-Strings definiert wird (`const char*`).
- Eine Klasse x wird in der Datei x`.h` deklariert, die Klasse heisst x`_c`, außerdem wird dort x`_t` als Zeigertyp für diese Klasse (also x`_c*`) deklariert. Die Implementierung der Methoden (sofern nicht `inline`) steht in x`.cpp`.

- Konstanten, die mit `#define` in x`.h` außerhalb der Klasse deklariert werden, beginnen mit dem Präfix X`_` (also wie üblich in Großbuchstaben). Insbesondere ist für jede Include-Datei sichergestellt, dass der Compiler ihre Deklarationen nur einmal verarbeitet, auch wenn sie ggf. mehrfach gelesen wird. Dazu wird in x`.h` gleich zu Anfang getestet, ob das Symbol X`_INCLUDED` schon definiert ist, und in diesem Fall der Rest der Datei übersprungen. Ansonsten wird das Symbol deklariert.
- Außerdem stellt jede Include-Datei sicher, dass Dateien, die verwendete Typen enthalten, automatisch auch gelesen werden. Man braucht also später in den Programmen nur die Include-Dateien für die direkt verwendeten Typen angeben. Zur Effizienzsteigerung teste ich vorher, ob X`_INCLUDED` schon definiert ist, das ist aber nicht wichtig.
- Es gibt eine Datei `ver.h`, in der Konstanten definiert sind, die man eventuell ändern möchte.
- Ich verwende in großen Maß Assertions (Zusicherungen, die zur Laufzeit geprüft werden, solange in `ver.h` das Symbol `VER_DEBUG` definiert ist. Als ich damit angefangen habe, war `assert.h` noch nicht Standard, und ich habe mir mein eigenes `check.h` gemacht (auch mit gewissen Unterschieden, z.B. einer extra Fehlermeldung, und einer Variante für `assert(false)`, die keine Compiler-Warnung erzeugt). Mein Haupt-Makro zum Test ist `CHECK(Cond,ErrMsg)`. Jede Klasse hat eine Methode `valid()`, die die Integrität des Objektes prüft, und im positiven Fall einen Null-Pointer liefert, im negativen Fall eine Fehlermeldung (ein String). Jede andere Methode der Klasse ruft `CHECK_VALID(Methodenname)` auf, um die Integrität des aktuellen Objektes zu sichern. Wenn man ein als Parameter übergebenes Objekt prüfen will, geht das mit `CHECK_PAR(Obj,Methodenname)`.
- Es ist zu beachten, dass Assertions (sowohl mit `assert.h` als auch mit meinem `check.h`) nur während des Debuggens aktiviert sind. Fehler, die keine Programmierfehler sind, müssen dagegen immer geprüft werden. Das betrifft auch eine Zerstörung der Datenstrukturen in den Blöcken der Datenbank-Datei (es könnte ja jemand die Datei mit Word öffnen und wieder abspeichern . . .). Während wir die Wiederherstellung im Fehlerfall nicht implementieren, soll eine Beschädigung der Daten möglichst frühzeitig und sicher erkannt werden.

Die grundlegenen Include-Dateien und ein Beispiel-Makefile stelle ich auf der Webseite der Übung zur Verfügung.

a) Legen Sie eine Datei `ver.h` an, und definieren Sie darin eine Konstante `VER_BLOCKSIZE` als 8192 (8 KByte). Natürlich könnte man für solche Optionen auch eine Konfigurationsdatei machen, die immer beim Start des Programms gelesen wird. Dann müsste das Programm nicht neu compiliert werden, wenn man die Blockgröße ändern will. Allerdings kann man diese Größe dann nicht in Typ-Deklarationen verwenden. Sie können sich als Startpunkt auch die Datei `ver.h` von der Übungs-Webseite kopieren, sie enthält bereits die geforderte Konstante.

b) Legen Sie eine Datei `btype.h` mit einem Aufzählungstyp `btype_t` für Blocktypen an. Man braucht mindestens die Werte

- `BTYPE_EMPTY` für nicht belegte Blöcke,
- `BTYPE_FILE_HEADER` für den ersten Block der Datei mit Basisdaten der Datenbank (z.B. Freispeicher-Verwaltung) und
- `BTYPE_REL_DATA` für Blöcke mit Relationen-Daten.

Ich habe außerdem den ersten Wert (0) als `BTYPE_INVALID` definiert (für nicht initialisierte Blöcke), bin mir aber im Moment nicht sicher, ob man diesen Wert braucht. Auch die Datei `btype.h` steht schon auf der Webseite.

Falls man das Programm später erweitern will, würde es noch weitere Arten von Blöcken geben, z.B. Blöcke für B-Baum Indexe, und Blöcke für Relationen mit variabler Zeilenlänge. Das werden wir aber in den Hausaufgaben zu dieser Vorlesung nicht mehr schaffen.

c) Definieren Sie nun eine Klasse `block_c` für Datenbank-Blöcke (in der Datei `block.h`). Diese Klasse wird später Unterklassen für die verschiedenen Arten von Blöcken haben. In meinem Programm sind Objekte der Klasse `block_c` genau `VER_BLOCKSIZE` Bytes groß (sie enthalten insbesondere ein Array von Zeichen für die eigentlichen Daten, mit einer `union` überlagert mit `short` und `int`-Werten). Die Unterklassen fügen dann neue Methoden hinzu, aber keine neuen Attribute. Das bedeutet natürlich, dass man die Datenstrukturen in den Blöcken auf relativ tiefer Implementierungsebene aufbauen muss (es werden Konstanten und Makros für die Berechnung von Offsets im Daten-Array definiert).

   Man könnte natürlich auch anders vorgehen, und einen Block zunächst wesentlich kleiner als 8 KByte sein lassen. Die Lese- und Schreib-Routinen werden aber immer 8 KByte zwischen Hauptspeicher und Platte bewegen. Eventuell braucht man ein Container-Objekt der vorgegeben Größe, in das man die tatsächlich benötigten Daten als kleineres Objekt hineinlegt (mit einem Typ-Cast oder einer Union).

d) Die Klasse `block_c` muss mindestens eine Methode `type()` haben, die den Typ des jeweiligen Blocks liefert (also einen Wert vom Typ `btype_t`).

e) Es ist aber zu empfehlen, dass man ausserdem die Korrektheit des Blocks mit einer Methode `ok()` prüfen kann (das Ergebnis könnte eine Fehlermeldung als String sein, oder ein Null-Zeiger im positiven Fall — wie die oben erwähnte Methode `valid()`, nur dass `ok()` auch im Nicht-Debug-Modus existiert). Man könnte z.B. in jeden Block am Anfang und eventuell auch am Ende ein bestimmtes Bitmuster schreiben ("Magic Number"), an dem erkannt wird, dass es ein Datenblock von diesem Programm ist.

f) Es wird später nützlich sein, wenn man auch die Nummer des Blockes in der Datei im Block-Objekt speichert. Dafür habe ich die Methode `block_no()` zum Abfragen der Blocknummer und `set_block_no()` zum Setzen der Blocknummer vorgesehen. Die Blocknummer ist ein 32-Bit Wert (Datenbanken können groß sein). Man könnte sie `unsigned` machen, muss dann aber später öfters mit Typ-Inkompatibilitäten kämpfen.

   Ich habe einen Typ `bno_t` (in `bno.h`) für Blocknummern definiert, aber möglicherweise ist das übertrieben.

g) Ein üblicher Trick, um nur partiell geschriebene Blöcke zu erkennen, ist es, ein Bitmuster am Anfang und am Ende eines Blockes abzulegen, das bei jedem Schreibvorgang invertiert wird. Die Bitmuster müssen dann immer gleich sein. Wenn Sie das machen wollen, können Sie eine Methode `prepare_write()` vorsehen, die diese Bitmuster invertiert, und ein Mal vor jedem Schreibvorgang aufgerufen wird.

h) Machen Sie sich außerdem Gedanken über die Fehlerbehandlung. Was wollen Sie z.B. tun, wenn die Datenbank-Datei nicht eröffnet werden kann, oder ein eingelesener Block die Integritätsbedingungen nicht erfüllt? (Sie brauchen diese Gedanken nicht aufzuschreiben oder abzugeben. Wir unterhalten uns in der nächsten Übung über diesen Punkt.)

Im wesentlichen ist das Ziel dieses Aufgabenblattes, sich über den Programmierstil klar zu werden (meinen oder Ihren eigenen), ein paar allgemeine Grundlagen zu legen, und dann die Oberklasse für Blöcke zu deklarieren. Sie können entweder nur die Klassendeklaration einsenden, oder ein Archiv (z.B. `tar`) mit allen Dateien Ihres Projektes. Sie sollten bereit sein, Ihre Klassendeklaration und Ihren Programmierstil in der Übung zu erklären, und sich aktiv an der Diskussion um die Fehlerbehandlung zu beteiligen.