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

15. Dezember 2015

Datenbanken II B: DBMS-Implementierung
— Hausaufgabe 8A —

In dieser Aufgabe soll das Implementierungs-Projekt fortgesetzt werden.

a) Messen Sie bitte die Laufzeit Ihres Puffer-Managers für die Datei `refstring.txt` auf dem Rechner, den wir auch für den Oracle-Server benutzen. Verwenden Sie dabei die Version aus Blatt 7, die schon Speicherplatz für die maximal 2000 Blöcke reserviert, aber noch nicht wirklich Blöcke aus einer Datei läd. Geben Sie außerdem die Anzahl Hits und Misses für den Puffer an.

b) Nun soll der Puffer-Manager so erweitert werden, dass er tatsächlich Blöcke aus einer Datei liest bzw. in diese schreibt. Legen Sie dazu zunächst eine Klasse `file_c` in den Dateien `file.h` und `file.cpp` an. Die Dateien werden für den Puffer-Manager über kleine positive Zahlen (IDs) angesprochen. Es wäre z.B. möglich, dass ein anderes Modul des Programms (das Sie nicht schreiben müssen), eine Art "Control File" wie bei Oracle liest, und dann alle Dateien einer Datenbank unter festen IDs eröffnet. Wenn man ROWIDs/TIDs hat, die zwischen Dateien verweisen (von einem Index in eine Relation), dann muss die ID einer Datei natürlich stabil bleiben.

   Außerdem hat diese Methode den Vorteil, dass Sie eventuell systemabhängige Details in `file.cpp` verstecken können, aber die Schnittstelle `file.h` nicht unbedingt Include-Dateien für Streams etc. beinhalten muss. Wenn Sie so vorgehen wollen, würde `file.cpp` eine Hilfsklasse enthalten, die "die eigentliche Arbeit macht", und z.B. ein Attribut für den jeweiligen Stream, File Pointer, File Descriptor, oder File Handle hat. Von diesen Objekten muss man dann ein kleines Array verwalten, das die IDs in die eigentlichen Daten abbildet.

   Die Klasse `file_c` soll folgende statischen Komponenten an der öffentlichen Schnittstelle anbieten:

   - Eine Konstante `FILE_MAXID` für die maximale ID einer Datei (z.B. 20).
   - Eine statische Methode `open(id, name)`, um die Datei mit Namen `name` unter der ID `id` zu öffnen. Die Datei muss zum Lesen und Schreiben in binärem Modus eröffnet werden. Es soll `true` zurückgegeben werden, wenn das Öffnen erfolgreich war, und `false`, falls nicht (z.B. Datei existiert nicht oder die Zugriffsrechte reichen nicht aus). Unter der gleichen ID darf keine Datei aktuell offen sein.
   - Eine statische Methode `filename(id)`, die den Dateinamen zur Datei mit ID `id` liefert. Die Datei muss natürlich vorher geöffnet worden sein.
   - Eine statische Methode `size(id)`, die zu einer offenen Datei mit ID `id` die Dateigröße in der Anzahl Blöcke liefert. Die Dateigröße aller mit dieser Klasse

verarbeiteten Dateien muss immer ein ganzzahliges Vielfaches der Blockgröße `block_c::BLOCK_SIZE` sein. Auch hier muss die Datei vorher geöffnet worden sein.

- Eine statische Methode `read(id, blockno, ptr)`, mit der man einen Block mit Nummer `blockno` aus der Datei mit ID `id` liest. Das Ergebnis (`BLOCK_SIZE` Bytes) soll an die Speicheradresse `ptr` (Zeiger auf ein Objekt der Klasse `block_c`) geschrieben werden. Es soll `true` geliefert werden, wenn die Operation erfolgreich war, und `false`, falls nicht (z.B. Datei zu kurz). Natürlich muss die Datei offen sein.

- Eine statische Methode `write(id, blockno, ptr)`, mit der man den Block an der Hauptspeicher-Adresse `ptr` in die Datei mit ID `id` schreibt, und zwar an die Block-Position `blockno`. Auch hier muss die Datei offen sein.

- Eine statische Methode `sync(id)`, die sicherstellt, dass alle geschriebenen Blöcke tatsächlich auf die Platte gelangt sind (und nicht noch in einem Puffer der Bibliothek oder des Betriebssystems auf ein späteres Schreiben warten). Die Datei `id` muss auch hier geöffnet sein. Falls Sie die Pufferung des Betriebssystems nicht in den Griff bekommen, sollten Sie wenigstens mit `flush()` den Puffer der Stream-Bibliothek leeren.

- Eine statische Methode `close(id)`, die die Datei mit ID `id` wieder schließt.

- Eine statische Methode `create(id, name, numblocks)`, die eine Datei mit Namen `name` anlegt und darin `numblocks` leere Blöcke schreibt. Die Datei soll anschließend geöffnet sein, so wie mit `open(id, name)`.

- Eine statische Methode `extend(id, numblocks)`, die die Datei mit ID `id` um `numblocks` leere Blöcke verlängert. Die Datei muss vorher bereits geöffnet sein.

Beachten Sie bitte, dass wenn Sie den Betriebsssytem-Aufruf `read` in einer Klasse verwenden wollen, die selbst eine Methode `read` hat, Sie `::read` schreiben müssen. Die klassische UNIX/POSIX-Funktion zum Setzen der Lese/Schreibposition in einer Datei ist `lseek`. In der C stdio Bibliothek ist es `fseek`. Wenn Sie C++ Streams verwenden, informieren Sie sich über `seekg` und `seekp`. Eventuell ist auch folgende Quelle hilfreich:

[http://stackoverflow.com/questions/15670359/fstream-seekg-seekp-and-write]

Die Größe der Datei können Sie bestimmen, indem Sie auf das Ende positionieren (d.h. Position 0 relativ zum Ende) und dann die aktuelle Position (relativ zum Anfang) bestimmen (`lseek` liefert diese Position als Ergebniswert, für C++ Streams gibt es `tellg` und `tellp`).

Ich habe in meinem Programm die UNIX File Deskriptoren verwendet. Die Methode `sync()` hatte folgende Aufrufe:

```
        // Call OS sync function:
        int n = -1;
#if _POSIX_C_SOURCE >= 199309L || _XOPEN_SOURCE >= 500
        n = fdatasync(FD);
#elif  _BSD_SOURCE || _XOPEN_SOURCE || _POSIX_C_SOURCE >= 200112L
        n = fsync(FD);
#else
        CHECK_IMPOSSIBLE("file_c::sync: No fdatasync and no fsync");
#endif

        // Check return value:
        if(n != 0) {
                alert_c::err_sync(Filename, strerror(errno));
                return false;
        }
```

In meinem Programm dient die Klasse `alert_c` der Speicherung von Fehlermeldungen. Sie sind nicht verpflichtet, Fehlermeldungen zu programmieren. Es könnte sich aber lohnen, darüber nachzudenken. Sie dürfen die Schnittstelle der Klasse ggf. erweitern. Das Makro `CHECK_IMPOSSIBLE` erzeugt in meinem Programm eine Laufzeit-Fehlermeldung, ähnlich einem fehlgeschlagenen `assert`.

c) Schreiben Sie nun ein Testprogramm, das eine Datei mit 50000 Blöcken anlegt. Schreiben Sie in jeden Block die Block-Nummer (ggf. müssen Sie dazu die Klasse `block_c` erweitern, oder eine Unterklasse davon anlegen). In einem zweiten Durchlauf des Programms messen Sie wieder die Zeit für die Zugriffe aus `refstring2.txt`, wobei jetzt die Blöcke tatsächlich gelesen werden. Kontrollieren Sie jeweils, dass die Blocknummer stimmt. Es ist wichtig, dass das Anlegen der Datei und die Zeitmessung in zwei getrennten Programmdurchläufen geschieht (so dass zumindest die Puffer des eigenen Programms leer sind, bei entsprechendem zeitlichen Abstand eventuell auch die des Betriebssystems).

— Hausaufgabe 8B —

Gegeben sei eine Datenbank mit den Tabellen

- EMP(EMPNO, ENAME, SAL, DEPTNO→DEPT, MGR°→EMP)
- DEPT(DEPTNO, DNAME, LOC)

Dies ist eine etwas vereinfachte Version der klassischen Oracle-Beispieldatenbank. Nehmen Sie an, die Angestellten-Tabelle EMP enthält 10000 Zeilen, und die Abteilungs-Tabelle DEPT 25 Zeilen, und die Angestellten pro Abteilung seien gleichverteilt. Ein Vorgesetzter (MGR) habe durchschnittlich 10 Untergebene. Nehmen Sie weiter an, dass 10 Datensätze pro Block gespeichert werden (bei beiden Tabellen). Es sei die folgende Anfrage gegeben:

```
SELECT   EMPNO, SAL, DNAME
FROM     EMP E, DEPT D
WHERE    E.DEPTNO = D.DEPTNO
AND      E.MGR = 7839
ORDER BY SAL
```

Beschreiben Sie, wie man die Anfrage auswerten kann, wenn man folgende Indexe hat:

d) Keinen.

e) EMP(EMPNO) und DEPT(DEPTNO) (beide natürlich UNIQUE).

f) Wie e), zusätzlich EMP(MGR).

g) Wenn Sie sich einen Index aussuchen dürften, auch einen mit mehreren Attributen, welchen würden Sie wählen? Dabei soll es nur darum gehen, dass die obige Anfrage möglichst schnell abgearbeitet wird.