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

17. Juni 2013

Datenbanken II B: DBMS-Implementierung — Projektbeschreibung —

Es soll ein kleines DBMS programmiert werden, das über eine Programmierschnittstelle benutzt werden kann. In der ersten Ausbaustufe ist kein Mehrbenutzerbetrieb und keine Anfragesprache geplant. Die Schnittstelle erlaubt es zunächst nur, eine Relation vollständig zu durchlaufen, oder einen Datensatz mit einer gegebenen ROWID zu finden. Mögliche Datentypen für Spalten sind `int`, Strings, und ROWID (dadurch kann man Verzeigerungsstrukturen aufbauen). Natürlich soll alles leicht erweiterbar sein.

Typ `str_t`

Im folgenden wird `str_t` als String-Typ verwendet, also `const char *` in C++. Damit können String-Literale für Argumente von diesem Typ übergeben werden.

Klasse `*_c`, Pointer-Typ `*_t`

Die Namen aller hier definierten Klassen enden in `_c`. In C++ braucht man häufig Zeiger auf Objekte, der Zeigertyp für die Klasse *X*`_c` heißt im folgenden immer *X*`_t`.

Klasse `db_c` (Initialisierung, Beendigung)

- `static init (str_t alert_file, bool print_err = false)`
  Initialisierung der Datenbank-Bibliothek. Für `alert_file` kann ein Nullzeiger übergeben werden, dann wird kein Fehlerprotokoll geschrieben. Falls `print_err` wahr ist, werden Fehlermeldungen auf den Standard-Fehler-Kanal `cerr` ausgegeben. Als Fehler zählen Dinge, die der programmierer nicht beeinflussen kann (z.B. Datei existiert nicht oder Inhalt ist zerstört, oder kein Hauptspeicher mehr, etc.). Bei Programmierfehlern, also auch fehlerhaften Aufrufen der hier angegebenen Methoden, darf der Programmlauf mit "assertion failed" abgebrochen werden. Falls das Symbol `VER_DEBUG` nicht definiert ist, darf alternativ ein normaler Fehler gemeldet werden. Wer möchte, darf bei Fehlern außerdem eine Exception auslösen.
- `static checkpoint()`
  Schreiben aller veränderten Datenblöcke.
- `end()`
  Beendigung der Datenbank-Bibliothek. Impliziert automatisch einen Checkpoint.

Klasse `file_c` (Datenbank-Dateien, enthalten Relationen)

Als Benutzer muß man sich zunächst ein oder mehrere Objekte der Klasse `file_c` anlegen, um Zugriff auf Daten-Dateien zu bekommen (oder sie am Anfang anzulegen). Folgende Methoden sind für den Benutzer interessant:

- `file_c(str_t filename, int ID)` (Konstruktor)
- `bool create(int blocks)`
- `bool open()`
- `bool close()`

Klasse `rel_c` (Relation/Tabelle)

- `rel_c(file_t file, str_t name)` (Konstruktor)
- `bool create()`
  Prüft, dass die Datei nicht bereits eine Relation mit dem Namen enthält, und legt dann so eine Relation an. Vor dem Aufruf von `create()` müssen `col_c`-Objekte für die Spalten angelegt werden.
- `bool open()`
  Prüft, ob die Datendatei eine Relation mit diesem Namen enthält, und läd ggf. Zugriffsdaten der Relation. Außerdem wird geprüft, dass die mit zugeordneten `col_t`-Objekten definierten Spalten existieren. Die Relation kann natürlich weitere Spalten haben, außer denen, für die `col_c`-Objekte erzeugt wurden.

Klasse `col_c` (Spalte/Attribut)

- `col_c(rel_t rel, str_t name)` (Konstruktor)
  Spalten können nur erzeugt werden, solange das `rel_c`-Objekt noch nicht geöffnet wurde.

Eventuell Subklassen `col_int_c` u.s.w. je nach Datentyp.

Klasse `rscan_c` (Cursor/Scan/Iterator über Tupel einer Relation)

- `rscan_c(rel_t)` (Konstruktor)
- `bool_t open()`
- `bool fetch()`
  Muß auch vor dem ersten Tupel aufgerufen werden. Liefert `false`, falls Ende der Relation erreicht.
- `int int_val(col_t col)`
  Zugriff auf Spaltenwert des aktuellen Tupels. Natürlich muß die Spalte vom Typ `int` sein.

- `str_t str_val(col_t col)`
- `tid_t tid_val(col_t col)`
- `tid_t current()`
- `bool_t close()`

Es ist implementierungsabhängig, was passiert, wenn ein Tupel eingefügt wird, während ein Scan offen ist. Das Programm darf aber auf keinen Fall abstürzen. Die Implementierung kann nur festlegen, ob das neue Tupel gesehen wird oder nicht, oder ob beide Fälle möglich sind (undefiniert).

Klasse `tbuf_c` (Puffer für ein Tupel einer Relation)

- `tbuf(rel_t rel)` (Konstruktor)
- `insert()`
  Erzeugt neues Tupel (initialisiert mit 0 bzw. leerem String in allen Spalten).
- `bool load(tid_t tid)`
  Findet Tupel zu gegebener TID/ROWID.
- `int int_val(col_t col)`
  Zugriff auf Spaltenwert des Tupels im Puffer.
- `str_t str_val(col_t col)`
- `tid_t tid_val(col_t col)`
- `int int_update(col_t col, int new_val)`
  Update eines Spaltenwertes.
- `str_t str_update(col_t col, str_t new_val)`
- `tid_t tid_update(col_t col, tid_t new_val)`
- `free()`
  Tupel muß nicht mehr im Puffer gehalten werden. Erst danach ist neues `insert()` bzw. `load()` möglich.

Beispiel

Das Schema könnte in einer Header Datei stehen:

```
file_c db("data.dbf", 1);
rel_c stud(&db, "Studenten");
col_int_c sid(&stud, "SID");
col_str_c vname(&stud, "Vorname");
col_str_c nname(&stud, "Nachname");
```

Einmalig müssten danach `create`-Aufrufe folgen:


```
db.create(100);
stud.create();
```

Ansonsten könnten im Programmcode `open`-Aufrufe benutzt werden:

```
db.open();
stud.open();
```

Durchlaufen und Ausgeben der Relation würde dann so aussehen:

```
rscan_c stud_scan(&stud);
while(stud_scan.fetch()) {
stud_scan.open();
while(stud_scan.fetch()) {
    cout << stud_scan.int_val(&sid) << " ";
    cout << stud_scan.str_val(&vname) << " ";
    cout << stud_scan.str_val(&nname) << "\n";
}
stud_scan.close();
```