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

27. Mai 2013

Datenbanken II B: DBMS-Implementierung — Hausaufgabe 6 —

Definieren Sie eine Klasse `seg_c` für Segmente, d.h. Folgen von Blöcken in einer Datei. Segmente dienen u.a. zur Implementierung von Relationen. Die Klasse soll mindestens die folgenden Methoden haben:

- `seg_c(file_t file)`
  Der Konstruktor legt zunächst ein Segment-Objekt im Hauptspeicher an, das noch nicht mit einem bestimmten Segment in der Datei verknüpft ist.

- `bool open(int id)`
  Diese Funktion prüft, ob es ein Segment mit der Nummer `id` in der Datei gibt. Falls ja, werden ggf. Daten über das Segment (z.B. Start-Blocknummer) in das Objekt geladen, und es wird `true` geliefert. Falls es kein Segment mit der Nummer `id` gibt, oder ein Fehler beim Dateizugriff auftritt, wird `false` geliefert.

- `bool create(int size)`
  Diese Funktion legt ein neues Segment in der Datei an, und ordnet automatisch eine freie `id` zu. Es werden auch mindestens `size` Blöcke reserviert und dem Segment zugeordnet. Es ist möglich, das Segment etwas größer als `size` Blöcke zu machen (wenn der Speicher in größeren Einheiten verwaltet wird). Falls das Segment erfolgreich angelegt wurde, wird `true` geliefert. Ansonsten (z.B. nicht genügend Speicher vorhanden, keine Segment-`id` mehr frei, Fehler beim Dateizugriff) wird `false` geliefert.

  Falls die Datei nicht genügend freie Blöcke hat, könnte eventuell später automatisch eine "AUTOEXTEND"-Funktion des Datei-Objektes aufgerufen werden. Dazu müßte die Klasse `file_c` entsprechend erweitert werden (man muß wählen können, ob ein AUTOEXTEND erwünscht ist, und in welchen Schritten bis zu welcher Maximalgröße). Für diese Hausaufgabe ist aber nicht verlangt, dass Sie eine solche Funktion implementieren: Es reicht, einfach `false` zurückzugeben. Die hier vorgeschlagene Schnittstelle läßt es allerdings nicht zu, verschiedene Fehlerquellen zu unterscheiden, so dass der Aufrufer nur "ins Blaue hinein" versuchen könnte, die Datei im Fehlerfall zu verlängern. Es ist aber natürlich möglich, dem Benutzer diese Aufgabe zu übertragen: Er sieht ja die Fehlermeldung und könnte entsprechend reagieren.

  Es wäre eventuell auch günstig, die Klasse `file_c` um eine Methode zu erweitern, die die größte benutze Blocknummer liefert (und ggf. eine Methode, um diesen Wert zu setzen).

  Nach Aufruf von `create()` soll das Segment in geöffnetem Status sein.

- `bool extend(int blocks)`
  Diese Funktion soll das Segment um `blocks` Blöcke verlängern. Wieder wird `true` geliefert, wenn die Verlängerung erfolgreich ausgeführt wurde, und `false` sonst. Zum Aufruf von `extend()` muß das Segment vorher geöffnet (oder erzeugt) worden sein.

- `id()`
  Diese Funktion liefert die Nummer dieses Segmentes (bzw. `-1` falls das Segment noch nicht erzeugt oder eröffnet wurde).

- `drop()`
  Diese Funktion soll das Segment löschen. Hinweis: Zuerst sollte die Funktion `delete` heißen, aber das ist ein reserviertes Wort in C++. Falls es zu aufwendig wird, dürfen Sie diese Funktion in der ersten Ausbaustufe weglassen.

- `buf_t get_block(int n)`
  Diese Funktion soll einen Puffer liefern, in dem der `n`-te Block des Segmentes gepinnt ist. Falls das nicht möglich ist, soll ein Nullzeiger zurückgegeben werden. Der Aufrufer ist dafür verantwortlich, zu gegebenem Zeitpunkt `unpin()` für den Puffer aufzurufen.

- `int abs_bno(int n)`
  Diese Methode soll eine segment-relative Blocknummer `n` in eine absolute Blocknummer (d.h. datei-relative) umrechnen. Die Methode ist optional.

- `bool close()`
  Diese Methode soll das Segment schließen. Man kann anschließend wieder `open()` oder `create()` aufrufen. Es soll dagegen als Fehler zählen, `open()` oder `create()` für ein Segment im geöffneten Status aufzurufen.

Es ist zulässig, die Anzahl der Segmente pro Datei, also den Bereich der möglichen IDs zu begrenzen. Sie können dafür z.B. eine Konstante `VER_MAX_SEGMENTS` definieren. Pro Datei sollen mindestens 32 Segmente möglich sein, besser 100. Es ist zulässig, die Anzahl von sequentiell gespeicherten Stücken (Extents), aus denen sich ein Segment zusammensetzt, zu begrenzen. Minimal sollten drei Stücke möglich sein.

Sie dürfen die Klasse `block_c` erweitern, z.B. um einen Verkettungszeiger (Nummer des nächsten Blocks). Außer dem Headerblock der Datei gehören alle Blöcke zu Segmenten (oder sind noch unbenutzt). Es wäre vermutlich günstig, wenn man von jedem Block die Nummer seines Segmentes abfragen könnte.

Eventuell können Sie die Liste der Segmente im Headerblock der Datei unterbringen (dieser Block ist ja bisher wenig genutzt). Je nach Implementierung brauchen Sie aber auch eine extra Freispeicherliste. Es ist zulässig, die Größe der Dateien zu begrenzen, aber 1 GByte sollte möglich sein.

Zur Implementierung von Relationen wird voraussichtlich im Segment mit der ID 1 fest eine Liste der Relationen abgespeichert werden, und im Segment mit ID 2 eine Liste von Spalten (Data Dictionary). Diese IDs werden natürlich über Konstanten definiert. Wenn Sie z.B. Segment 1 für die Freispeicherliste benötigen, wäre das möglich. Es steht Ihnen selbstverständlich frei, die Zählung mit 0 zu beginnen.

— Hausaufgabe 6B —

Definieren Sie eine Klasse `sscan_c` für Scans/Iteratoren über den Blöcken eines Segmentes. Die Klasse soll folgende Methoden haben:

- `sscan_c(seg_t seg)`
  Konstruktor. Das übergebene Segment muß in geöffnetem Status sein.
- `bool open()`
  Scan öffnen bzw. auf den Anfang zurücksetzen.
- `buf_t fetch()`
  Diese Methode holt den nächsten Block des Segmentes. Sie muß auch für den ersten Block aufgerufen werden. Es wird ein Puffer geliefert, in dem der Block gepinnt ist (bzw. ein Nullzeiger, falls das Ende des Segmentes erreicht ist, oder ein anderer Fehler auftritt).

  Der Aufrufer sollte wohl unterscheiden können, ob das Segmentende normal erreicht ist, oder ein Fehler aufgetreten ist. Wenn Sie mit Exceptions arbeiten, ist das natürlich gegeben. Ansonsten könnte man eine extra Methode dafür einführen, oder beim `close()` `false` zurückliefern, falls es beim Scan einen Fehler gegeben hat.

  Beim `fetch()` sollte ein eventuell zuvor gepinnter Block automatisch freigegeben werden (mit `unpin()`. Falls der Aufrufer den Block wirklich noch braucht, steht es ihm frei, selbst noch einmal `pin()` aufzurufen (Hier ist allerdings ungünstig, dass `pin()` eine statische Methode ist, also eine neue Suche des Blockes stattfinden muß — falls Sie beim `pin()` nicht ein optionales "Hint"-Argument vorgesehen haben. Eventuell könnte man eine zusätzliche Methode `repin()` oder `pin_too()` vorsehen.)
- `int current_bno()`
  Dies liefert die aktuelle Nummer des Blockes im Segment, also die Anzahl von Malen, die `fetch()` aufgerufen wurde. Sie können sich aussuchen, ob Sie die Zählung mit `0` beginnen. Die gelieferte Zahl muß aber mit `get_block()` von `seg_c` funktionieren.

  Wenn Ihre Segmentverwaltung Blöcke nie bewegt, ist es auch möglich, die absoluten Blocknummern in der Datei zu verwenden. Diese Funktion dient dazu, ROWIDs bei einem Full Table Scan zu konstruieren.
- `close()`
  Schließen des Scans. Wenn Sie keine zusätzliche Methode zur Fehlerabfrage haben, sollten Sie `false` zurückliefern, wenn beim Scan irgendwann ein Fehler aufgetreten ist.
- `bool jumpto_bno(int bno)`
  Für bestimmte Anwendungen (z.B. Mergejoin) wäre es günstig, eine Methode zu haben, mit der man die Nummer des nächsten mit `fetch()` zu holenden Blockes festlegen kann. Vermutlich werden wir solche Joins in diesem Semester nicht mehr implementieren, die Methode ist daher optional.