UNIVERSITÄT DES SAARLANDES
Fachbereich 14– Informatik
Prof. Dr.–Ing. G. Weikum
Dipl.–Inform. A. Kraiß

Datenbanksysteme WS 1998/99

Übung 6 — Beispiellösung zu Aufgaben 1, 2

Aufgabe 1: LFU Verdrängungsstrategie

Betrachten Sie das folgende Pufferersetzungsverfahren, das unter dem Namen **LFU** (Least Frequently Used) bekannt ist.

> *Es wird jeweils diejenige Seite im Puffer ersetzt, die seit ihrer Einlagerung in den Puffer am wenigsten häufig referenziert wurde.*

Zur Implementierung dieses Verfahrens genügt ein einfacher Referenzzähler pro Seite im Puffer. Überlegen Sie sich die Vor– und Nachteile dieses Verfahrens.

Lösung:

Dieses Verfahren beinhaltet keinen Alterungsmechanismus für die Seiten im Puffer. Eine sehr häufig referenzierte Seite bleibt solange im Puffer, bis es Seiten gibt, die noch häufiger referenziert wurden. Im Laufe der Zeit füllt sich der Puffer mit häufig referenzierten Seiten. Ersetzungsopfer werden immer unter den wenigen neu eingelagerten Seiten gesucht. Neu eingelagerte Seiten haben dann nie mehr die Chance, selber häufig referenziert zu werden. LFU ist deshalb für Datenbanken mit dem sich ständig ändernden Referenzverhalten der Transaktionen nicht brauchbar.
Notwendig ist ein Alterungsmechanismus für die bisher schon im Puffer befindlichen Seiten und ein Schutzmechanismus für neu eingelagerte Seiten.

Zusatzinformation für Interessierte:

GCLOCK (Generalized Clock) ist eine Variante der LFU–Strategie, bei der ein Alterungsmechanismus für jede Seite im Puffer eingebaut ist. Durch die Spezifikation eines Anfangsalters kann ein frühzeitiges Verdrängen verhindert werden. Diese modifizierte Variante eignet sich besser für Datenbanken. Genaueres zum Thema Pufferverwaltung kann in *W. Effelsberg, T. Härder: Principles of Database Buffer Management, ACM Transactions on Database Systems, 9(4), 1984* nachgelesen werden.

Anhand des folgenden Beispiels kann man das schlechte Verhalten von LFU im Vergleich zu GCLOCK beobachten:

Wir betrachten einen Puffer mit einer Kapazität von 3 Seiten. Beim LFU–Verfahren wird bei jeder Referenzierung einer Seite deren Zähler um 1 erhöht. Bei GCLOCK sei angenommen, daß eine neu eingelagerte Seite einen Zählerwert von 3 bekommt. Bei jeder Referenzierung wird der Zähler um 1 erhöht und alle zwei Zeiteinheiten um 1 reduziert, wenn die Seite in der Zwischenzeit nicht referenziert wurde. Für jeden Pufferrahmen bedeutet die erste Zahl die Nummer der eingelagerten Seite und die Zahl in der Klammer der aktuelle Stand des Zählers. Referenzen, die zu einem Plattenzugriff führen, sind jeweils fett gedruckt.

Zeitpunkt	logische Referenz	LFU			GCLOCK		
		Rahmen 1	Rahmen 2	Rahmen 3	Rahmen 1	Rahmen 2	Rahmen 3
1	1	**1 (1)**	–	–	**1 (3)**	–	–
2	2	1 (1)	**2 (1)**	–	1 (3)	**2 (3)**	–
3	3	1 (1)	2 (1)	**3 (1)**	1 (2)	2 (3)	**3 (3)**
4	1	1 (2)	2 (1)	3 (1)	1 (3)	2 (2)	3 (3)
5	2	1 (2)	2 (2)	3 (1)	1 (3)	2 (3)	3 (2)
6	1	1 (3)	2 (2)	3 (1)	1 (4)	2 (3)	3 (2)
7	2	1 (3)	2 (3)	3 (1)	1 (4)	2 (4)	3 (1)
8	4	1 (3)	2 (3)	**4 (1)**	1 (3)	2 (4)	**4 (3)**
9	3	1 (3)	2 (3)	**3 (1)**	1 (3)	2 (3)	**4 (3) raus, 3 (3) rein**
10	4	1 (3)	2 (3)	**4 (1)**	**1 (2) raus, 4 (3) rein**	2 (3)	3 (3)
11	3	1 (3)	2 (3)	**3 (1)**	4 (3)	2 (2)	3 (4)
12	4	1 (3)	2 (3)	**4 (1)**	4 (4)	2 (2)	3 (4)
...							

Zum Zeitpunkt 9 haben bei GCLOCK alle Seiten den Zählerwert 3. Man kann bei Zeitpunkt 9 wieder nach verschiedenen sekundären Strategien ein Opfer aussuchen. Selbst in dem für unsere Referenzfolge ungünstigsten Fall sind bereits nach Zeitpunkt 10 die richtigen Seiten im Puffer. Für LFU ist zum Zeitpunkt 5 klar, daß die Seiten 1 und 2 für längere Zeit im Puffer bleiben. Nur noch eine Seite, die mehrmals hintereinander referenziert wird, kann diese zwei Seiten verdrängen. Viel realistischer ist aber eine abwechselnde Referenzfolge, bei der — wie im Beispiel gezeigt — die Seiten aus dem Pufferrahmen 3 immer wieder ausgelagert werden müssen.

Aufgabe 2: Implementierung einer LRU-Pufferverwaltung

Sie sind an der Implementierung eines Datenbanksystems beteiligt und sollen die Teilkomponente *ReplacementManager* der Pufferverwaltung implementieren. Der ReplacementManager soll drei Operationen (in C–Syntax)

```
PAGE*   fix(unsigned pageId);
        /* Dient zum Fixieren einer Seite im Puffer. Damit wird die Seite im virtuellen
           Speicher direkt adressierbar. */
void    unfix(PAGE* pageToUnfix);
        /* Dient zum Freigeben einer Seite im Puffer. Damit wird die Seite verdrängbar */
PAGE*   getReplacementVictim();
        /* Dient zum Ermitteln des nächsten zu verdrängenden Opfers. */
```

zum Fixieren und Freigeben einer Seite im Puffer bzw. zum Ermitteln des nächsten Verdrängungskandidaten anbieten. Die Datenstruktur PAGE beschreibt den Aufbau einer Seite im Puffer und enthält bisher die Identifikation der Seite, eine Markierung, ob sie seit der Einbringung in den Puffer modifiziert wurde und den Datenbereich der Seite. Die Deklaration für *PAGE* in C–Syntax sieht folgendermaßen aus:

```
typedef struct
{
        unsigned    pageId;                 /* ID der Seite im Puffer   */
        bool        dirty;                  /* Markierung, ob Seite
                                               modifiziert wurde*/
        char        data[PAGE_SIZE];        /* Datenbereich             */
} PAGE;
```

Ihr Chef gibt Ihnen vor, daß Sie eine LRU–Verdrängungsstrategie implementieren sollen und eine Seite durch Aufruf von *fix* als referenziert (used) gilt. Eine referenzierte Seite soll solange unverdrängbar sein, bis für sie ein *unfix* aufgerufen wird. Die Operation *getReplacementVictim* soll die nach LRU–Reihenfolge als nächste zu verdrängende verdrängbare Seite zurückliefern. Die ausgesuchte Seite soll für den ReplacementManager dann als verdrängt gelten, d.h. nach Aufruf von *getReplacementVictim* ist die zurückgelieferte Seite logisch nicht mehr im Puffer.

Der Pufferaufbau ist bereits teilweise durch die folgende Struktur *BUFFER* festgelegt. Der Puffer steht Ihrem Modul als globale Variable *TheBuffer* zur Verfügung.

```
struct BUFFER
{
        unsigned    capacity;       /* Kapazität des Puffers (# Seiten)     */
        unsigned    freePages;      /* Anzahl freier Seiten im Puffer               */
        PAGE page[MAX_PAGES];       /* Seiten im Puffer                     */
} TheBuffer;
```

Zusätzlich wird Ihnen (glücklicherweise) die Datenstruktur HashTable zur Verfügung gestellt, zusammen mit den Operationen

```
PAGE*       getEntry( HashTable h, unsigned pageId );
            /* Liefert Eintrag innerhalb der Tabelle h an der Stelle 'pageId'   */
void        insertEntry( HashTable h, unsigned pageId, PAGE* value );
            /* Fügt Eintrag (pageId,PAGE*) in Tabelle h ein                     */
void        removeEntry( HashTable h, unsigned pageId );
            /* Entfernt Eintrag mit Schlüssel 'pageId' aus Tabelle h            */
```

zum Ermitteln, Einfügen und Entfernen von Einträgen [Seiten–Id, Zeiger auf Seite] in einer Hash–Tabelle h.

a) Die Operation *fix* muß dafür sorgen, daß die angeforderte Seite im Puffer fixiert wird, d.h. daß sie unverdrängbar wird. Erst wenn unfix aufgerufen wird, kann die Seite aus dem Puffer verdrängt werden. Erklären Sie, warum die Seite nach Aufruf von *fix* nicht aus dem Puffer verdrängt werden darf.

Lösung zu a)

Eine angeforderte Seite darf nach Aufruf von *fix* nicht aus dem Puffer verdrängt werden, da ihre virtuelle Speicheradresse im Puffer an das anfordernde Modul übergeben wird, das potentiell Änderungen auf dieser Seite ausführen will (z.B. Einfügen/Ändern/Löschen eines Tupels) bzw. den Inhalt der Seite weiterverarbeiten wird. Erst wenn *unfix* aufgerufen wird signalisiert das Modul, daß die Modifikation/Verarbeitung beendet ist und daß dadurch die Seite nicht länger direkt referenziert wird.

b) Überlegen Sie sich eine effiziente Datenstruktur für die Verwaltung des Puffers mit einer LRU–Verdrängungsstrategie. Sie dürfen dazu die vorhandenen Datenstrukturen *BUFFER* und *PAGE* beliebig erweitern.

Lösung zu b)

Wir verwenden für die Implementierung der LRU–Strategie eine doppelt verkettete Liste von Pufferseiten, die einen Stack realisiert. Jede referenzierte Seite wird aus der aktuellen Position im Stack 'herausgezogen' und nach oben verlagert. Dadurch wandern unreferenzierte Seiten im Stapel nach unten. Die potentiellen Verdrängungsopfer finden sich dann ganz unten im Stapel. In der Datenstruktur BUFFER führen wir hierzu jeweils einen Zeiger auf den Beginn der Liste (Oberes Ende des Stapels) und auf das Ende der Liste (Unteres Ende des Stapels) ein. Für das effiziente Ermitteln der virtuellen Adresse aufgrund der Seiten–ID verwenden wir eine Hash–Tabelle. Somit sieht BUFFER nach unserer Erweiterung folgendermaßen aus:

```
struct BUFFER
{
        unsigned    capacity;           /* Kapazität des Puffers in Anzahl Seiten   */
        unsigned    freePages;          /* Anzahl freier Seiten im Puffer
*/
        PAGE        page[MAX_PAGES]; /* Seiten im Puffer                            */
        HashTable bufferContents;       /* Tabelle [pageId, Zeiger auf PAGE] mit
                                           Inhalt des Puffers                       */
        PAGE*       stackTop;           /* Zuletzt referenzierte Seite              */
        PAGE*       stackBottom;        /* Am längsten unreferenzierte Seite        */
} TheBuffer;
```

Die Datenstruktur PAGE muß zusätzlich eine Markierung enthalten, ob die Seite gerade fixiert ist oder ob sie verdrängt werden darf. Außerdem muß die doppelte Verkettung vorgenommen werden können, d.h. es muß jeweils ein Zeiger für die Vorgänger–Seite und die Nachfolger–Seite hinzugenommen werden. Somit ergibt sich folgende Struktur:

```
typedef struct
{
        unsigned    pageId;             /* ID der Seite im Puffer           */
        bool        dirty;              /* Markierung, ob Seite
                                           modifiziert wurde                */
        bool        isFixed;            /* TRUE, wenn Seite fixiert ist     */
        PAGE*       previousPage;       /* Vorgänger–Seite im Puffer        */
        PAGE*       followingPage;      /* Nachfolger–Seite im Puffer       */
        char        data[PAGE_SIZE];    /* Datenbereich                     */
} PAGE;
```

c) Geben Sie in C– oder PASCAL–ähnlicher Syntax eine Implementierung für die Operationen *fix*, *unfix* und *getReplacementVictim* an. Die Implementierung soll dabei die in a) entwickelten Datenstrukturen verwenden.

Lösung zu c)

Die Operation *fix* muß nun die referenzierte Seite zunächst im Puffer auffinden. Dazu wird das Inhaltsverzeichnis des Puffers (contents) verwendet. Danach muß die Seite als fixiert markiert werden und schließlich an das obere Ende des Stapels verlagert werden. Es ergibt sich folgende Implementierung:

```
/********************************** FIX ********************************/
PAGE*   fix( unsigned pageId )
{
    PAGE*   pageToFix;
    /*********************************************************/
    /*  Seite ermitteln und als 'fixed' markieren            */
    /*********************************************************/
    pageToFix= getEntry( &TheBuffer.bufferContents, pageId );
    pageToFix–>isFixed= TRUE;
    /*********************************************************/
    /*  Seite aus LRU–Kette ausklinken                       */
    /*********************************************************/
    if (pageToFix–>previousPage != NULL)
    {
        pageToFix–>previousPage–>followingPage= pageToFix–>followingPage;
    }
    if (pageToFix–>followingPage != NULL)
    {
        pageToFix–>followingPage–>previousPage= pageToFix–>previousPage;
    }
    else
    {
        /*****************************************************/
        /*  Seite ist am unteren Ende des Stapels            */
        /*****************************************************/
        if (pageToFix–>previousPage != NULL)
        {
            TheBuffer.stackBottom= pageToFix–>previousPage;
        }
    }
    /*********************************************************/
    /*  Seite an die Spitze des Stapels setzen               */
    /*********************************************************/
    pageToFix–>previousPage= NULL;
    pageToFix–>followingPage= TheBuffer.stackTop;
    TheBuffer.stackTop= pageToFix;
    return pageToFix;
}


/******************************** UNFIX ********************************/
void   unfix( PAGE* pageToUnfix )
{
    pageToUnfix–>isFixed= FALSE;
}
```



```
/********************** GETREPLACEMENTVICTIM ********************/
PAGE*   getReplacementVictim()
{
    PAGE*   victim= TheBuffer.stackBottom;
    while(victim != NULL)
    {
        if (! victim–>isFixed)
        {
            /*****************************************************/
            /*  Seite aus LRU–Kette ausklinken                   */
            /*****************************************************/
            if (victim–>previousPage != NULL)
            {
                victim–>previousPage–>followingPage= victim–>followingPage;
            }
            if (victim–>followingPage != NULL)
            {
                victim–>followingPage–>previousPage= victim–>previousPage;
            }
            else
            {
                if (victim–>previousPage != NULL)
                {
                    TheBuffer.stackBottom= victim–>previousPage;
                }
            }
            /*****************************************************/
            /*  Seite aus Inhaltsverzeichnis entfernen           */
            /*****************************************************/
            removeEntry( &TheBuffer.bufferContents, victim–>pageId );
            return victim;    /* Kandidaten gefunden */
        }
        victim= victim–>previousPage;      /* nächsten Kandidaten untersuchen */
    }
    return victim;    /* keine Seite verdrängbar */
}
```

Das hier vorgestellte Verfahren zur Verwaltung eines LRU–Puffers hat im Gegensatz zu einer einfachen LRU–Tabelle konstanten Aufwand für das Verschieben einer Seite innerhalb der LRU–Reihenfolge.

d) Ist es denn sinnvoll, daß eine Seite in dem Moment als referenziert gilt, wenn *fix* aufgerufen wird? Welchen Nachteil hat diese Vorgehensweise?

Lösung zu d)

Eine Seite zum Zeitpunkt des *fix* als referenziert anzusehen hat den Nachteil, daß Seiten, auf denen lang andauernde Operationen ausgeführt werden, benachteiligt werden. Denn vom Zeitpunkt des *fix* bis zum *unfix* können nebenläufige Anforderungen im Puffer dazu führen, daß die Seite während der Ausführung der Operation innerhalb der LRU–Kette nach unten verlagert wird und somit unmittelbar nach Aufruf von *unfix* aus dem Puffer verdrängt werden könnte. Im Hinblick auf eine gute Puffernutzung sollte deshalb eine Seite nach Ausführung der Operation, d.h. nach dem *unfix*, als referenziert gelten und in der LRU–Kette nach oben verlagert werden.