Buffer Manager: Project 1 Assignment

Due: Feb 11, 2003

Introduction to Database Systems,
UC Berkeley Computer Science 186 Spring 2003

---

1 Overview of Project 1 - Buffer Manager

In this project you will add functionality to the backend server of PostgreSQL. You need to delve into the actual server code. In this first assignment we limit our investigation to one core module, the *buffer manager*. The actual coding for this project is minimal, but you will need to figure out what code to add and where, which is non-trivial! The major parts of this project are:

- Understanding different memory management strategies
- Examining, understanding and modifying existing code
- Measuring the effects of your code in a real system

Due to disk space constraints, we are providing you with a leaner PostgreSQL source tree. Some features (contributed utilities, regression tests, manual, tutorials and other language support) have been removed. You will not need these for this project, but you are free to download them if you have the space. Even with these modifications, you will not have extra space to store other files, so be sure to remove them before beginning [1].

Make sure your code runs smoothly on the x86 Solaris EECS instructional machines (rhombus, pentagon, and torus.cs.berkeley.edu) prior to submission since ALL grading will done on those. However, since all the software is freely available, you may find it easier to develop your code at home or on other machines you have access to. The instructions provided are designed to work with the EECS instructional machines and may not work as smoothly elsewhere. The TAs and Professor will only support EECS instructional machines.

2 Tasks and Steps

(1) Learn different buffer management strategies. (30%)

[1] You can allocate temp space via `/share/b/bin/mkhometmpdir`.
See `http://inst.eecs.berkeley.edu/cgi-bin/pub.cgi?file=/usr/pub/disk.quotas` for details.

(2) Compile and install PostgreSQL.
(3) Implement CLOCK. (40%)
(4) Compare performance of LRU, MRU and CLOCK. (30%)
(5) Submit.

Please spend some time to read this assignment completely before beginning.

3 Learn Different Buffer Management Strategies (30%)

The textbook describes LRU, MRU and CLOCK strategies in section 9.4.1. However, you may need to read the whole section of 9.4 to get a full image of buffer manager in DBMS.

As a first step, we present a small exercise to test your knowledge of the replacement strategies. For LRU, MRU, and CLOCK, you need to trace the buffer contents for the workload listed below. For your solutions, list each time that a memory access caused a "miss" in the buffer pool, and which (if any) page is evicted from the buffer pool to make room for the missing page. Assume time starts from T1 and moves forward by one on each page reference.

Solution format for this step: You must record your answers in a plain text file called `strategies`. The format of the file is two columns of text, separated by spaces. The file should begin with your measurements from LRU, with each row listing the time of a miss, a space, and the page (if any) evicted from memory. Then do the same for MRU and CLOCK in that order, leaving a blank line in between. For example:

Time	Page Removed	
T1		←LRU
T2	W	
T5	Z	
		(a blank line here)
T2		←MRU
T5	Z	
		(a blank line here)
T4		←CLOCK
T5	W	

Note that not each time is listed for each strategy, since not each time has a buffer miss. Note also that the second column will be blank when the buffer miss does not result in an eviction.

Assume there are 4 pages your buffer manager must manage. All four pages are empty to start. For each access the page is pinned, and then immediately unpinned [2]. The

[2] You cannot make this same assumption when writing the actual code. A page may be pinned for any length of time.

following table provides a list of page accesses at each time step. Based on this information, you should be able to generate the `strategies` file correctly.

Time	Page Read
T1	A
T2	B
T3	C
T4	A
T5	D
T6	E

Time	Page Read
T7	E
T8	A
T9	B
T10	F
T11	C
T12	G

Time	Page Read
T13	A
T14	B
T15	B
T16	D
T17	G
T18	F

Time	Page Read
T19	A
T20	F
T21	C
T22	A
T23	B
T24	G

4 Compile and Test PostgreSQL

Everything needed for this project is available at `~cs186/sp03/Hw1/hw1_pkg.tar.gz` on all instructional machines. To begin with, copy this package to your own directory and untar it with: `gtar -xzf hw1_pkg.tar.gz`[3].

You will see three subdirectories in your `Hw1/` directory, which are:

(1) `postgresql-7.2.2/`: Source code of PostgreSQL version 7.2.2. Little code was changed between this version and the actual version. The block size was changed to be smaller, in order to generate more I/Os. We added some scripts to help you compile your code. We also added some debugging/logging lines in the code.
(2) `src/`: In this directory you will find `freelist.c.mru`, which is our implementation of MRU. PostgreSQL ships with LRU, so our implementation of MRU is intended to provide you with an example of how to change the buffer manager's replacement policy. You will want to compare this code to `postgresql-7.2.2/backend/storage/buffer/freelist.c` to study the changes we made. You will also find `stub/` in this directory, which is the test harness code we will discuss later in Section 5.3.
(3) `exec/`: Some scripts you need to run your code.

Here are instructions for compiling PostgreSQL and updating your implementation to PostgreSQL.

(1) To completely re-compile and install PostgreSQL, `cd postgresql-7.2.2` and run `./compile.sh`.
(2) To only update your implementation to PostgreSQL and re-compile it, edit your own copy of `freelist.c` and `buf_internals.h` in the `src/` directory, `cd postgresql-7.2.2`, and run `./update.sh <your-freelist.c> <your-buf_internals.h>`.
NOTE: You should edit your own versions of `freelist.c` and `buf_internals.h` in the `src/` directory, and use the `./update.sh` command to install them.

[3] Remember to remove it if you are out of disk space.

We STRONGLY recommend that you do not modify the files in the `postgresql-7.2.2` directory directly! `update.sh` will use your files to replace related files in `src/backend/storage/buffer/` and `src/include/storage/`, and compile your code with PostgreSQL. The original LRU files have been copied for your reference as `*.c.lru` and `*.h.lru` in their original directories. Of course you need to pay attention to any errors produced by the compiler while running this script.

5 Implement CLOCK (40%)

5.1 How to implement?

LRU (Least Recently Used) is the most common policy used by operating and database systems. However, direct implementation of LRU is pretty expensive. So some systems use a policy called "CLOCK" instead. While "CLOCK" approximates the behavior of the LRU scheme, it is much faster.

The description of "CLOCK" algorithm is available in Section 9.4.1 of the textbook. We give a description here to assist your implementation.

To implement the "CLOCK" replacement policy, you need to maintain the following information. You are free to maintain additional information if you want. But the following two pieces of information are necessary.

(1) A *referenced* bit associated with each page. Each time a page in the buffer pool is done being accessed, the *referenced* bit associated with the corresponding frame should be set to 1 before the page is considered for replacement. In your implementation, you can simply set the *referenced* bit to 1 when you unpin a page and its *pin_count* goes to 0 – that is the first time that the the page is available to the replacement strategy.
(2) The "clock hand" (*current* in the textbook): varies between 0 and `NBuffers` - 1, where `NBuffers` is the number of buffers in the PostgreSQL buffer pool. Each time a page needs to be found for replacement, the search begins from from the *current* page and advances to '`current = (current + 1) % NBuffers`' if not found (i.e. in a clockwise cycle).

Each page in the buffer pool is in one of three states:

- pinned: *pin_count > 0*
- referenced: *pin_count = 0* and *referenced = 1*
- available: *pin_count = 0* and *referenced = 0*

To find a page for replacement, you start from the *current* page, repeat the following steps until you find a page or all pages are pinned. If:

- page[*current*] is pinned: advance *current* and try again.

- page[*current*] is referenced: set status to 'available', advance *current* and try again.
- page[*current*] is available: use this page and advance *current*.

5.2 Which files to modify?

The actual implementation of CLOCK is rather straightforward once you understand the existing code. *One hint we'd like to give here is that you don't need to bother with the LRU freelist queue. You can add and manage any new data structure you need without doing anything with the old ones.* The existing code is not extremely clear, but it is understandable. It may take you a few hours (or more) to understand it. Since understanding the existing code is a significant part of the assignment, the TAs and Professor will not assist you in understanding the existing code (beyond what we discuss here).

The actual buffer manager code is neatly separated from the rest of the codebase. It is located in `src/backend/storage/buffer` and `src/include/storage`. For CLOCK, we are interested in modifying only two files.

src/backend/storage/buffer/freelist.c , manages which pages are not pinned in memory, and which pages are eligible for replacement.

src/include/storage/buf_internals.h contains the definition for each buffer description (called `BufferDesc`). Most of the fields in `BufferDesc` are managed by other parts of the code, but for efficiency some fields are used to manage the buffer space while it is eligible for replacement (when it's on the existing LRU freelist).

Some initialization of the queue occurs in `src/backend/storage/buffer/buf_init.c`. However, you must do all your initialization in freelist.c.

You will modify these two files to change the implementation from the existing LRU queue to the CLOCK discipline. You may find that you do not need to edit both files or change many of the functions.

N.B.: You may need to declare global variables to make your code work. While this is generally poor programming style and you may have been taught to avoid it, it is the most natural solution given the existing structure of PostgreSQL. It's wise to use the C modifier `static` to limit the scope of your global variable to a single `.c` file.

Your code will be tested with both our test harness stub (30%, see description below) and running in PostgreSQL on a real dataset (10%). Unfortunately there is no partial credit for each test. Be sure it is correct. A bad buffer manager makes the rest of the system useless!

5.3 How to debug?

We suggest that you debug by running a debugger on the Postgres backend in standalone mode. You can following the instructions given during the C/GDB review

section. More information on using the backend in stand-alone mode is available at: `http://www.postgresql.org/idocs/index.php?app-postgres.html`

We suggest you following steps for debugging [4]:

(1) If you want to log which pages are loaded and which pages hit, add `#define CS186_HW1_DEBUG` in your `buf_interals.h`. The debugging code we added for you is in `src/backend/storage/buffer/bufmgr.c`, and you are welcome to read or even change it as you see fit. You can also use following function to add your own entries to the Postgres debugging output: `elog(DEBUG,<format>,<arg>)` – see `bufmgr.c` for examples [5].
(2) Update and compile your code using `update.sh` as described in section 4. You don't need to use '`gmake install`' to install it here!
(3) Prepare your data directory: run `./init.sh <DATADIR>` and `./pre.sh <DATADIR>` in `~/Hw1/exec` [6]. `./pre.sh` will create a database named "test" and two tables. NOTICE: the script will completely remove the `<DATADIR>`. So be careful.
(4) Run `src/backend/postgres` (in a debugger if you like). Make sure you are running this version of `postgres`, not the one in the class account directory! To start the backend server in stand-alone mode, run `src/backend/postgres -s -D <DATADIR> test`. It will use the data directories (and data) you prepared in the last step. The interface is directly to the backend, so psql features will not work. If your binary does not work correctly, it may corrupt the data, so be warned.
(5) Through the backend, you can directly type SQL statements, although the output is somewhat painful to read. To exit, type `CTRL-D`. In addition, the -s option will tell the server to print statistics at the end of each query, including the number of buffer hits! Finally, you can restrict the number of buffers PostgreSQL will use by adding the `-B <nbuffers>` [7] option so that even small queries generate buffer misses.

Test harness stub: A small test harness stub program is available in `~Hw1/src/stub/*`. Copy your versions of `freelist.c` and `buf_internals.h` to override the original ones in `src/stub`, and run `make` to compile the test harness stub. You may want to modify the stub to add additional test cases. The actual test stub we use for grading will be different; the one provided is only for your convenience (and to make sure your code will compile with our test stub)!

---

[4] This is not a requirement, of course, but we thought you'd find this method useful.
[5] Use (`postgres >& <filename>`) to redirect the debugging output to some file.
[6] The data directory you created in Homework 0 may not be compatible with this project, because we changed block size of each buffer smaller than the original PostgreSQL. So, make sure you create a new directory
[7] `<nbuffers>` should not be smaller than 16.

6 Compare performance of LRU, MRU and CLOCK (30%)

Here you will compare CLOCK with LRU and MRU. There are three scripts we prepared for you, all in `~/Hw1/exec`:

(1) `./init.sh <DATADIR>`: the script for you to init the data.
(2) `./pre.sh <DATADIR> [index]`: `[index]` is an optional flag that you will use in some experiments to create a table with an index on its primary key.
(3) `./run.sh <logfile> <DATADIR> <clock|lru|mru> <bufnum>`: Make sure you have compiled [8] and installed [9] your CLOCK version before you run it with the `clock` flag. The statistics of the query will be written in `<logfile>`. `<DATADIR>` is the same directory as (1) and (2). Use `clock`, `lru` or `mru` to assign which replace policy you want to use. `<bufnum>` is the number of buffers you want to use.

The `<logfile>` you get is like:

```
DEBUG: EXECUTOR STATISTICS
! system usage stats:
! ... ... ... ...
DEBUG: EXECUTOR STATISTICS
! system usage stats:
! ... ... ... ...
! postgres usage stats:
! Shared blocks: 842 read, 0 written, buffer hit rate = 86.31%
! ... ... ... ...
```

This is interpreted as follows: `842` is the actual read requests that had to be passed onto the disk (= total page requests for reads - buffer hits for reads), and `86.31%` is the hit rate of the query (= total buffer hits for reads / total page requests for reads).

The SQL scripts of creating tables and running queries are available at `~cs186/sp03/Hw1/*.sql`. You can read them if you're interested. The `query.sql` script basically does:
`SELECT * from S,R where R.fkey=S.key`.

You are asked to do two experiments.

(1) Prepare your data without index on S:
`./init.sh <DATADIR>`
`./pre.sh <DATADIR>`
Run `./run.sh` with `lru` or `mru` or `clock` and different `<bufnum>`, fill table below:

[8] Refer to Section 4 about how to compile
[9] Use "`gmake install`"

<bufnum>	250	270	290	310	330	350
LRU hit rate						
MRU hit rate						
CLOCK hit rate						

(2) Prepare your data with index on primary key of S:
`./init.sh <DATADIR>`
`./pre.sh <DATADIR> index`
Run `./run.sh` with `lru` or `mru` or `clock` and different `<bufnum>`, fill table below:

<bufnum>	250	270	290	310	330	350
LRU hit rate						
MRU hit rate						
CLOCK hit rate						

Please answer some questions based on experiment results you get:

(1) *For experiment 1, did each strategy perform as we described in class? Answer yes or no, and explain your answer.*
(2) *For experiment 2, what are the main differences compared with experiment 1? Explain the cause of these differences.*
(3) *Tables* `R` *and* `S` *are the same size. Can you tell the approximate <bufnum> used by* `R` *(or* `S`*)? How do you know that?*

A Hint: In answering the questions above, you may be interested to know that PostgreSQL uses 278 buffers to initialize your query (e.g. load system tables), and unpins these buffers before the query begins running.

7 Submit

You must submit at least 5 files (even if they are incomplete or unchanged):

(1) README - the members of your group (names and class accounts), what works, and what doesn't.
(2) strategies - answers to the first part, in the correct format!
(3) freelist.c.clock - CLOCK implementation
(4) buf_internals.h.clock - CLOCK implementation
(5) perf.txt - TEXT ONLY! Performance comparison, tables, answer to the question.

Make sure all the files above are in a directory called `~/Project1` in one of your group member's class account. `cd` to that directory, and type `submit Project1` to submit the files. Each group only needs to submit once. If there are multiple submissions, the last submission by any group member will be used for grading and the calculation of slip days.