COMP9315 19T2 DBMS Implementation

Prac Exercise 05

Buffer Pool Join Simulation

Aims

This exercise aims to get you to:

- implement a C program to simulate a range of buffer replacement policies
- evaluate experimentally how replacement policies and buffer pool size interact

Background

Database management systems rely heavily on the presence of numerous in-memory buffers to avoid excessive disk reads and writes. A point noted by Michael Stonebraker and others is that DBMSs know better than (e.g.) the underlying operating systems, the patterns of access to data on disk, and should be able to manage the use of in-memory buffers very effectively. Despite this, DBMSs still tend to rely on generic buffer replacement strategies such as "least recently used" (LRU) and "most recently used" (MRU).

In this exercise, we'll implement a simulator that allows us to look at a range of buffer pool settings and policies, to determine the best setting for dealing with one particular database operation: nested-loop join.

The nested-loop join is a particular method for executing the following SQL query:

```
select * from R join S
```

It can be described algorithmically as

```
for each page P in R {
   for each page Q in S {
      for each tuple A in page P {
         for each tuple B in page Q {
            if (A,B) satisfies the join condition {
               append (A,B) to the Result
            }
         }
      }
   }
}
```

When using a buffer pool, each page is obtained via a call to the `request_page()` function. If the page is already in the pool, it can be used from there. If the page is not in the pool, it will need to be read from disk, most likely replacing some page that is currently in the pool if there are no free slots.

If no buffer pool is used (i.e. one input buffer per table), the number of pages that will need to be read from disk is $b_R + b_R b_S$, where b_R and b_S are the number of pages in tables *R* and *S* respectively. Hopefully, accessing pages via a buffer pool will result in considerably less page reads.

Setup

For this exercise, you won't need PostgreSQL at all. However, there is a partly-completed version of the simulator available in the archive

```
/web/cs9315/19T2/pracs/p05/p05.tar
```

Un-`tar` this archive into a directory (folder) for this lab, and examine the files:

```
$ mkdir /my/directory/for/p05
$ cd /my/directory/for/p05
$ tar xf /web/cs9315/19T2/pracs/p05/p05.tar
$ ls
Makefile        bufpool.c       bufpool.h       joinsim.c
```

The file `joinsim.c` contains the main program which collects the simulation parameters, sets up the buffer pool, "runs" the nested-loop query and then displays statistics on the performance of the system. The `bufpool.*` files implement an ADT for the buffer pool. The `Makefile` produces an executable called `jsim` which works as follows:

```
$ ./jsim OuterPages InnerPages Slots Strategy
```

where

- *OuterPages* is the number of pages in the "outer" relation (R in the example above)
- *InnerPages* is the number of pages in the "inner" relation (S in the example above)
- *Slots* is the number of page slots in the buffer pool
- *Strategy* is the buffer replacement strategy, and can be one of
    - L ... least-recently used (page which was released earliest)
    - M ... most-recently used (page which was last released)
    - C ... cycling (cycles through the slots and picks the next available)

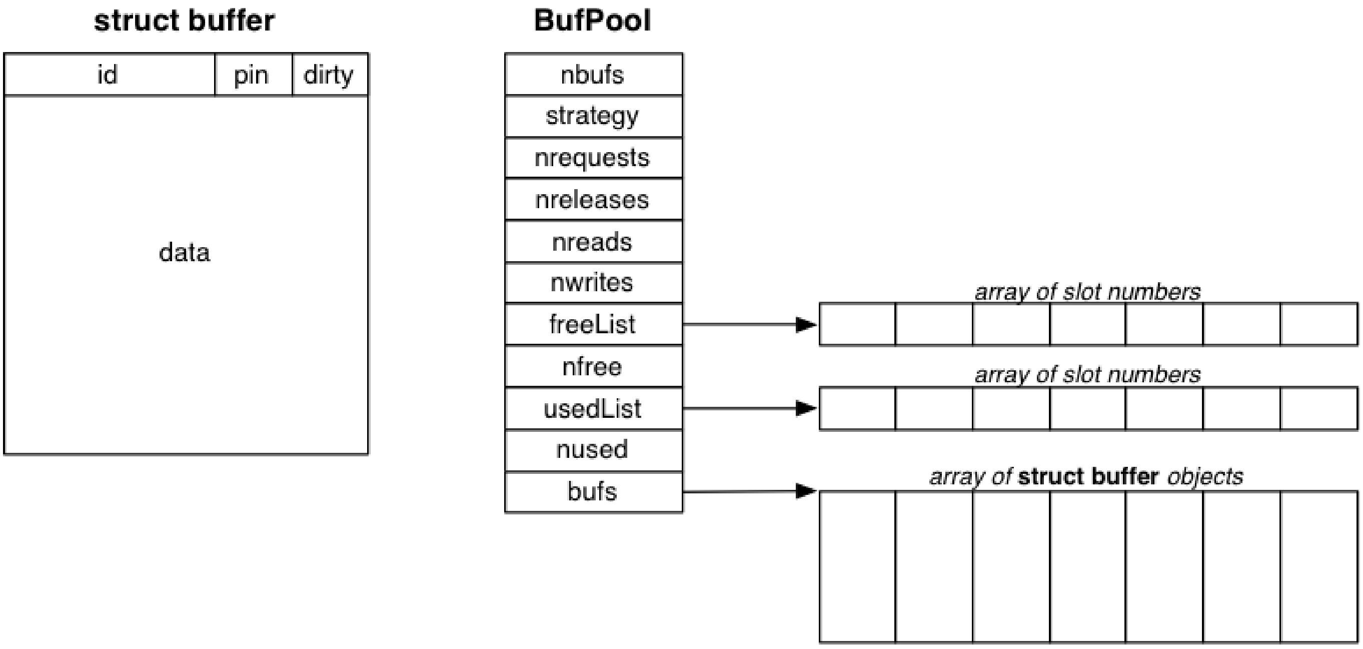
You can compile the program as supplied, but since part of the code is missing, you won't get sensible results. In some cases, you will even trigger one of the `assert()` tests.

To assist with testing, there is a compiled version of my solution available as:

```
/web/cs9315/19T2/pracs/p05/jsim0
```

Of course, being a pre-compiled binary, this will only work on a system compatible with those in the CSE labs.

What you should do now is read the code in `bufpool.c`. Start with the data structures, then look at the `initBufPool()` function to see how the data structures are set up. The following diagram may help with this:



The `nbufs` counter holds the total number of buffers in the system. This is set at initialisation time and determines the length of the three arrays. The `strategy` field contains one character represeting the replacement strategy (either `'L'`, `'M'` or `'C'`). The next four fields (`nrequests`, `nreleases`, `nreads` and `nwrites`) are statistics counters to monitor the buffer pool performance; they are manipulated correctly by the supplied code. The `freeList` is initially set to hold all of the slot numbers, since all pages are free. As pages are allocated, the free list becomes smaller and is eventually empty and stays empty; at this point, all of the slots are either in use or are in the `usedList`. The `usedList` holds slot numbers for buffers which have been used in the past, but which currently have a pin count of zero. These are the slots to be considered for removal if a new page, Finally, `nfree` and `nused` count the number of elements in the `freeList` and `usedList` respectively.

Next, look at the `request_page()` and `release_page()` functions (which capture the methods pretty much as described in lectures). Finally, look at the other functions used by `request_page()` and `release_page()`.

Exercise

Your task for this exercise is to complete the following functions in `bufpool.c`

getNextSlot(pool)

This function aims to find the "best" unused buffer pool slot for a `request_page()` call. It is called once it has been determined that (a) the requested page is not in the buffer pool, and (b) there are no free pages available. Thus, it needs to choose a slot from the used list; the "best" slot is determined by the replacement strategy for the buffer pool. If the used list is empty or if all slots have a pin count greater than zero, then `getNextSlot()` should return -1 (which will trigger errors higher up in the system). If a suitable slot is found, and if the page currently occupying that slot has been modified, the the page should be written out (note that we don't actually write anything, simply increment the `nwrites` counter for the buffer pool). Finally, `getNextSlot()` should clean out the chosen buffer, and return the index of the buffer slot to the caller. For the `'C'` strategy, set the "next available buffer" to the one immediately following the chosen buffer.

`makeAvailable(pool,slot)`

This function is called whenever the pin count on a slot reaches zero (meaning that this slot is now available for reuse). The function adds the slot number to the used list; where in the list it should be added is determined by the replacement strategy.

`removeFromUsedList(pool,slot)`

This function will be called when a previously occupied page has been chosen for use. It should search for the specified slot number in the used list and remove it from this list. Since the method depends on how the used list is managed, it is dependent on the replacement strategy.

Modify the above functions to achieve the specified behaviour. If you want to change other parts of the `BufPool` ADT (e.g. because you think my implementation is no good), feel free. If you come up with a much better solution than mine, let me know.

Once you've implemented the functions, test that they are behaving correctly, either by comparing the output to the output from `jsim0` or by thinking about the expected behaviour of the buffer pool.

Once you're satisfied that the functions are correct, investigate the behaviour of the buffer pool under various conditions. Consider variations on each of the following scenarios, where *N* is the number of buffers:

- $N = 2$
- $b_R + b_S < N$
- $b_R + b_S = N$
- $b_R + b_S > N$

Consider each case using each of the buffer replacement strategies. For each scenario, try to determine what will happen and then check your prediction by running `jsim`.

Challenge: PostgreSQL Clock-sweep Strategy

Modify the data structures to support the PostgreSQL clock-sweep buffer replacement strategy. Note that clock-sweep is not quite the same as the Cycle strategy used above. Once you've got it working, run the same set of tests that you ran for the other strategies and compare its performance.

End of Prac

Let me know via the forums, or come to a consultation if you have any problems with this exercise ... *jas*