Buffer Replacement Policies in Postgres

A comprehensive guide to implement and test a few BRP’s in Postgres

Table of Contents:

Significance of Buffer Replacement Policies

The DBMS can only operate on data that is in the RAM. The buffer manager hides from DBMS the fact that not all data is present in the RAM. Whenever a page is requested, it is checked to see if it is present in the buffer pool. If not, it has to be brought into the memory and an existing page has to be chosen for replacement if the buffer pool is full. The choice of which page to replace (the victim frame) is decided by the algorithm that is used by the buffer manager. Hence it is essential to use an efficient Page Replacement Algorithm to ensure that requested pages are made available with the least amount of page transfers between the disk and the memory, which is an expensive operation.

Why not use the Buffer Replacement Policy provided by the OS?

There are several reasons why a DBMS needs its own Buffer Replacement Policy. The files can span disks and the Buffer Manager still would need to work, which is not so if default OS implementation is

used. Different OS make use of different policies and hence there could be portability issues. Apart from these, the fact that the DBMS need greater control over how the pages are fetched and written to the disk also reinforces the need for a separate Buffer Replacement Policy for the DBMS. It needs control over pinning a page in a buffer pool, forcing a page to the disk, pre-fetch pages to improve performance of queries etc. Also, this makes it easier for the DBMS to recover in case of a crash.

General overview of our work

Most of the work is done on freelist.c. There are a couple of function that are of particular importance to us.

- StrategygetBuffer(): Called when Buffer Manager has to allocate a new buffer to load a page. It returns a free buffer (if available) or a buffer in which the page can be placed (selected based on the replacement policy used). We select a buffer such that it's pincount is 0 (i.e; it is not being referenced). Also, if the dirty bit is set (Page is modified), then we have to first write that page to the disk and then consider if for replacement.
- BufferUnpinned(): As the name suggests, this function is called when a buffer is unpinned and it's reference count reaches 0 and hence becomes a probable candidate for replacement
- Each buffer is represented by a BufferDesc structure located in buf_interhals.h. The structure is as shown:

```
typedef struct sbufdesc
{
        BufferTag tag; /* ID of page contained in buffer */
        BufFlags flags; /* see bit definitions above */
        uint16 usage_count; /* usage counter for clock sweep code */
        unsigned refcount; /* # of backends holding pins on buffer */
        int wait_backend_pid; /* backend PID of pin-count waiter */
        slock_t buf_hdr_lock; /* protects the above fields */
        int buf_id; /* buffer's index number (from 0) */
        int freeNext; /* link in freelist chain */
        LWLockId io_in_progress_lock; /* to wait for I/O to complete */
        LWLockId content_lock; /* to lock access to buffer contents */
} BufferDesc;
```

Of the above members, we are concerned primarily with usage_count and refcount. When usage_count is zero, the page can be reclaimed. If refcount is zero, the buffer is not pinned.

- initBufferPool(): This is the function in which buffers are created and Buffer Descriptors are initialized.

More on how to create the tables and test with different policies will be discussed soon.

Buffer Replacement Policies:

Clock Sweep:

Currently the algorithm being used by Postgres. Following is the code snippet:

```
if (BufferReplacementPolicy == POLICY_CLOCK)
{
        trycounter = NBuffers;
        for (;;)
        {
                buf = &BufferDescriptors[StrategyControl->nextVictimBuffer];
                if (++StrategyControl->nextVictimBuffer >= NBuffers)
```

```
                {
                        StrategyControl->nextVictimBuffer = 0;
                        StrategyControl->completePasses++;
                }
                LockBufHdr(buf);
                if (buf->refcount == 0)
                {
                        if (buf->usage_count > 0)
                        {
                                buf->usage_count--;
                                trycounter = NBuffers;
                        }
                        else
                        {
                                if (strategy != NULL)
                                        AddBufferToRing(strategy, buf);
                                return buf;
                        }
                }
                else if (--trycounter == 0)
                {
                        UnlockBufHdr(buf);
                        elog(ERROR, "no unpinned buffers available");
                }
                UnlockBufHdr(buf);
        }
}
```

As we can see, we maintain a circular queue of pages resident in the memory. A Referenced bit is used to track how often a page is referenced. It is set whenever a page is referenced. We only replace pages that haven’t been referenced for one complete cycle of the

“Clock”. Failing to find such a buffer will result in aborting the process requesting the page. Care has to be taken to ensure that dirty pages are not directly replaced without flushing changes to the permanent storage.

Least Recently Used: (LRU)

This policy was used in older versions of postgres. For each page in the buffer pool, we keep track of the last unpinned page. The Least recently used page is chosen as a candidate for replacement as long as it is not being referenced currently. If the page is being referenced, the next LRU page is checked to see if it is a suitable candidate for replacement. Following is the code snippet:

```
if (BufferReplacementPolicy == POLICY_LRU)
        {
                buf = StrategyControl->firstUnpinned;
                while (buf != NULL) {
                        LockBufHdr(buf);
                        if (buf->refcount == 0) {
                          resultIndex = buf->buf_id;
                          break;
                        } else {
                          UnlockBufHdr(buf);
                          buf = buf->next;
                        }

                }
                if (buf == NULL) {
               UnlockBufHdr(buf);
                        elog(ERROR, "no unpinned buffers available");
                }

}
```

If none of the buffers are available for replacement, buf will be NULL and we abort the process requesting the page. Clock is relatively better as compared to LRU in that it finds an old page, but not necessarily the oldest.

Once we saw the implementation of Clock in the newer postgres versions and LRU in the older one, we decided to merge them into one file and then test to see how it works. Once we got it working, we made slight changes to the LRU code to make it MRU.

Most Recently Used (MRU):

MRU is almost similar to LRU but it selects the Most Recently used page and checks if it is a suitable candidate for replacement.

Here's the code snippet for MRU.

```
if (BufferReplacementPolicy == POLICY_MRU)
{
                                buf = StrategyControl->lastUnpinned;
                                 while (buf != NULL) {
                                          LockBufHdr(buf);
                                          if (buf->refcount == 0) {
                                            resultIndex = buf->buf_id;
                                            break;
                                          } else {
                                            UnlockBufHdr(buf);
                                            buf = buf->previous;
                                          }
                                 }
                                 if (buf == NULL) {
                                UnlockBufHdr(buf); //p added 10/24
```

```
                elog(ERROR, "no unpinned buffers available");
            }
        }
```

Explanation is similar to LRU, except for the fact that MRU page is chosen for replacement.

2Q:

When we were searching for other buffer replacement policies to implement, we came across 2Q. This buffer replacement policy overcomes the drawback associated with LRU and Clock replacement policies. And that is that buffer pages that get accessed only once must wait an entire clock cycle before being replaced. In 2Q, we manage 2 separate queues, one FIFO queue and another LRU Queue.

Here's the pseudocode:

```
unpin(buf) {
    if buf is on the Am queue then
        put buf on the front of the Am queue
    else if buf is on the A1 queue then
        remove buf from the A1 queue
        put buf on the front of the Am queue
    else
        put buf on the front of the A1 queue
    end if
}
```

```
find_free_slot(p){
    if there are free slots available then
        put p in a free page slot
    else if size(A1) >= thres OR Am is empty /* we set thres = floor(B/2) for this
homework */
        delete from the tail of A1
        put p in the freed page slot
    else
        delete from the tail of Am
        put p in the freed page slot
    end if
}
```

Here’s the code snippet we found online:

```
if (BufferReplacementPolicy == POLICY_2Q)
                        {
                                int thres = NBuffers/2;
                                int sizeA1 = 0;
                                volatile BufferDesc *head = StrategyControl->a1Head;
                                while (head != NULL) {
                                        head = head->next;
                                        sizeA1++;
                                }
                                if (sizeA1 >= thres || StrategyControl->lastUnpinned == NULL) {
                                        buf = StrategyControl->a1Head;
                                        while (buf != NULL) {

                                         if (buf->refcount == 0) {
                                                resultIndex = buf->buf_id;
                                                next = buf->next;
                                                previous = buf->previous;
                                                //adjust neighbors
                                                if (next != NULL) {
                                                 if (previous != NULL)
                                                {
                                                previous->next = next;
                                                next->previous = previous;
                                                 } else {
                                                        next->previous = NULL;
                                                        StrategyControl->a1Head = next;
                                                 }
                                                } else if (previous == NULL) {
                                                 StrategyControl->a1Head = NULL;
                                                 StrategyControl->a1Tail = NULL;
                                                } else {
```

```
                    StrategyControl->a1Tail = previous;
                    previous->next = NULL;
                  }
                  buf->next = NULL;
                  buf->previous = NULL;
                  break;
                } else {
                  buf = buf->next;
                }
              }
              if (buf == NULL) {
                elog(ERROR, "no unpinned buffers available");
              }
}
```

All the code snippets along with the original files, perl scripts, results etc can be found at the following link:

https://github.com/SourabhShenoy/PostgreSQL/tree/master/Postgres%20Files

Testing:

Now we will look at how to create tables, add random data, select buffer size and buffer policy and also how to use explain-analyze to understand how many shared buffers are read/hit. Along with that, we will also explain the result for one Buffer replacement policy for one buffer size. The same explanation will hold good for others as well.

As a first step, either type the above code snippets or copy paste the freelist.c and related files (from the link given above) into your postgres

source code and replace the original file (You may want to back it up, just in case).

Once that is done, rebuild and reinstall postgres by typing :

```
make

make install
```

Next, start the psql client and create a table with say, one column of Varchar type. We will be filling this table with a million+ rows of random data through a perl script.

```
Create table table_name (random_string text);
```

The script is as follows:

```
#!/usr/bin/perl -w
use strict;
print generate_random_string() . "\n" for 1 .. 10_000_000;
exit;
sub generate_random_string {
my $word_count = 2 + int(rand() * 4);
return join(' ', map { generate_random_word() } 1..$word_count);
}
sub generate_random_word {
my $len = 3 + int(rand() * 5);
my @chars = ( "a".."z", "A".."Z", "0".."9" );
my @word_chars = map { $chars[rand @chars] } 1.. $len;
return join '', @word_chars;
}
```

You can obtain this code from insertdata.pl from the same link. Now that we have the perl script ready to generate a million random data, we need to find a way to copy the output of this script to the table.

To be able to do so, first install the perl DBI module by typing the following commands in Ubuntu:

```
$ wget http://search.cpan.org/CPAN/authors/id/T/TI/TIMB/DBI-1.625.tar.gz
$ tar xvfz DBI-1.625.tar.gz
$ cd DBI-1.625
$ perl Makefile.PL
$ make
$ make install
```

Add this line to pg_hba.conf before you start using perl with postgres:

```
# IPv4 local connections:
host    all         all         127.0.0.1/32          md5
```

The above steps can be skipped, but is recommended to do so in case you are planning to use perl in conjunction with postgres. Anyway, now redirect the output of the perl script to a .lst file.

Perl insertdata.pl > data.lst

test=#\copy table_name from 'data.lst';

This will generate ten million random strings of varied length and copy it to the table you created. You may create one more table using the same method.

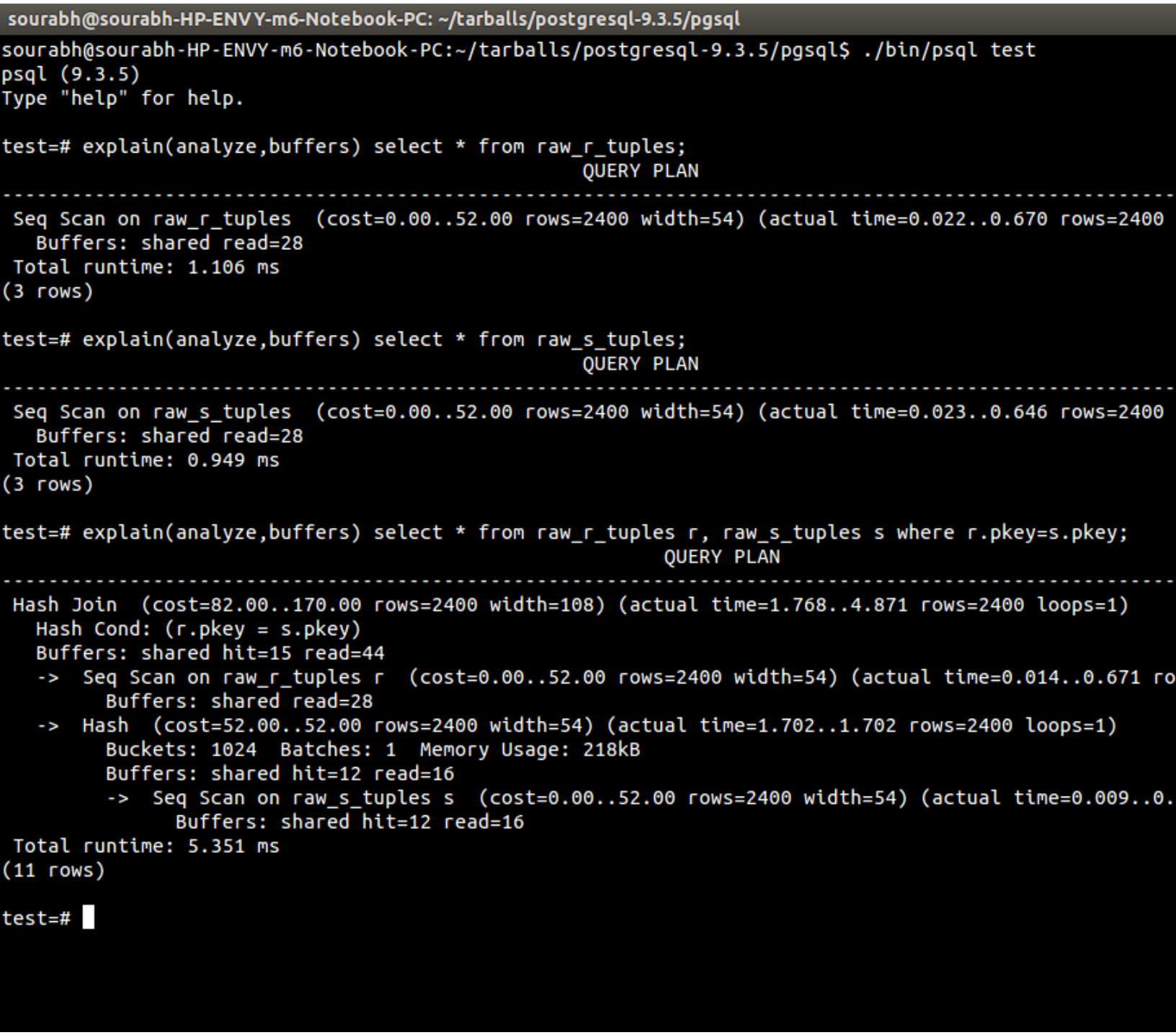
Now that we have installed postgres with the modified code and have the tables and data ready, it's time to start the postgres server by specifying the buffer size. But before that, we need to go to freelist.c and select the appropriate buffer replacement policy. A variable *BufferReplacementPolicy* has been used to select the Buffer Replacement Poilcy that will be used. This is however, a temporary solution as we are still finding a way to select this parameter through the command line at the time of starting the server. The variable can take one among the following 4 values: POLICY_CLOCK, POLICY_LRU, POLICY_MRU and POLICY_2Q.

Next, let us start the postgres server as follows:

```
./bin/pg_ctl start -D $HOME/pgsql/data -l lru64.log -o "-p 11111 -B 64
-N 1 -o '-te -fm -fh' --buffer-replacement-policy=lru --
autovacuum=off"
```

The –l option is used to generate a log file that will contain the outputs. We can check the last few lines of the log file by using the "tail" command. –p option is to specify the port number and –B is to specify the buffer size. We have tried with buffer sizes 16, 32, 64 and 96 for each of the Buffer Replacement policies. Autovacuum is a background program that may disrupt buffer operations and hence we must turn it off to get accurate results.

Now that we have seen how to start the server with different buffer sizes and to select the appropriate policy, let us look at a sample output (Shown in the next page).



The above picture is for 2Q page replacement policy for buffer size = 96.

We are performing three operations. One select on each of the table and one involving a join a join between the two tables

The Explain Analyze tool in postgres can be made use of to know the Query Execution Plan devised by postgres. It also has an optional parameter “Buffer” that displays the shared buffer usage (read/hit).

Just prepend your query with the explain (analyze, buffers) and then have a look at the number of buffers that are read and the number of buffers that are hit. Ideally, we would like to have high number of buffer hits (i.e; less reads from memory). In the third query where we perform a join, we can see the various steps that are taking place (Hash Join, Seq Scan, Hash Join, Seq Scan) and the number of buffers that are hit/read each time. Also, we can try the same with buffers of different size and also for different Page Replacement Policies.

Note: All the code snippets, files, perl scripts, results etc can be found on the following github link:

https://github.com/SourabhShenoy/PostgreSQL/tree/master/Postgres%20Files