CSCD43 DATABASE SYSTEM TECHNOLOGY

Assignment 1

Assigned January 21

Due Tuesday February 9

Weight: 10% of your final grade

Buffer Management

1 Introduction

In this assignment, you will modify the PostgreSQL 7.4.13 Buffer Manager. Before starting this assignment, you should have already downloaded, compiled and installed PostgreSQL 7.4.13. The goal of this assignment is to **understand** and **evaluate** various memory management strategies in a database system. The actual coding for this assignment is minimal, but you will be modifying existing source code in a very large codebase; therefore, it may be a bit challenging.

Start early!

2 Experimental setup

2.1 Data Generation and loading:

The table we will be working with can be created using the following SQL statement:

```
CREATE TABLE Data (
    ID INTEGER PRIMARY KEY,
    A INTEGER,
    B INTEGER,
    C INTEGER);
```

1. You **have to write a script** to generate a 5000-line file from which the table will be uploaded. In addition, you will have to make sure that:

   - the `ID` column is unique, and
   - the numbers inserted in the other columns are uniformly-random generated between 1-100.

   Each **line** in this file should be of the form:

   ```
   <id>, <A>, <B>, <C>
   ```

   An **example** of such file is the following:

   ```
   1,56,89,23
   2,45,78,45
   ...
   ```

2. Upload this file into the table `Data` using the `COPY` command in `psql`. Your data should now be ready. **Suggestion**: you can check the correctness of your data using some queries.

Assignment 1

2.2 Queries

1. **Write** 10 queries of the form:

```
SELECT COUNT(*)
FROM Data
WHERE <some range condition on A,B,C>;
```

   An **example** of such a query is below:

```
SELECT COUNT(*)
FROM Data
WHERE A > 20 AND B < 40;
```

2. Store these 10 queries in a file `ScanQueries.sql`, as follows:
   - each query is stored in a separate line, and
   - there is a blank line at the end of the file.

   These queries will be executed with sequential scans.

3. **Write** 10 queries of the form:

```
SELECT COUNT(*)
FROM Data
WHERE <some small range condition on ID>;
```

   An **example** of such a query is below:

```
SELECT COUNT(*)
FROM Data
WHERE ID > 7 AND ID < 20;
```

4. Store these 10 queries in a file `IndexScanQueries.sql`, as follows:
   - each query is stored in a separate line, and
   - there is a blank line at the end of the file.

   These queries should probably be executed with index scans.

5. Before proceeding further,
   - **try** executing these queries in PostgreSQL , and
   - **check**, using the `EXPLAIN` command in `psql`, that they are indeed executed using `Sequential Scans` or `Index Scans`, as specified.

2.3 Evaluation of the queries

By now, you should have the data and the queries to run on it.
To **run** the queries on a **backend**, and **extract the buffer hit rate**, you should run PostgreSQL in a standalone backend mode, using the command `postgres` as follows. Note that this is the easiest way to run a set of queries and extract "interesting" information from the backend.

```
cat <queryfile.sql> | postgres -B <numbuffers> -D <datadir> -d <debug-level> -s <databasename>
```

The output of the `postgres` command contains lots of information. Of interest to you is the *buffer hit rate* for each query. In order to make sure you

- understand its output, and
- can extract appropriate information,

we recommend you to run different queries.
The flags on the `postgres` command are explained below:

- The `-B` flag specifies the size of the buffer cache in blocks.
- The `-D` flag refers to the data directory.
- The `-d` flag sets the debug level. This flag will force the backend to print out useful statistics.
  - For this assignment, setting the debug-level to 1 or 2 should be enough.
- The `-s` flag in conjunction with the debug flag results in buffer hit rates being printed in the output.

For more information on `postgres`, you can use `man postgres`.
Note: in the evaluation section, you will have to vary the number of buffers, and to printe the hit rates for different configurations.
An **example** of the command above is as follows:

```
cat ScanQueries.sql | postgres -B 20 -D ~/data/ -d 1 -s mydatabase
```

Now, you may hack!

3 Implementation

The version of PostgreSQL you are using for this assignment uses the `LRU` buffer replacement policy. For this assignment you are required to implement the `LFU` (Least Frequently Used) and `MRU` (Most Recently Used) replacement policies. Consult your favourite textbook to find out the details of these policies.

You can find the **buffer manager code** in the directories

```
src/backend/storage/buffer
src/include/storage/
```

The files which will most likely need the most are:

- `freelist.c`
  - it manages pages which are not pinned in memory and are eligible for replacement.
- `bufmgr.c`
  - it defines the interface used by the rest of PostgreSQL to access the memory buffer.
- `buf_init.c`
  - it handles initializations of the buffer manager data structures.
- `buf_internals.h`
  - it defines the data structures used by the rest of the buffer manager.

You should be able to complete the entire assignment by modifying just these files.

We **recommend** that, before you start modifying the files to implement the new replacement policies, you save a copy of these files elsewhere. Note that you will need to keep separate copies of the 4 files for implementing the `LFU` and `MRU` policies.
Each time you make changes to the files, you have to compile and install PostgreSQL again. This can be done using the following commands:

```
$gmake clean
$gmake uninstall
$gmake
$gmake install
```

If you haven't modified any header files since you last compiled the code, you may avoid the `gmake clean` command. Also, you do not need to create the database again each time you recompile and reinstall PostgreSQL .

3.1 Tips on how to deal with such code

- **Do not** delete any old `LRU` code in the files. It is much easier to just comment it out and add a marker (for instance, `BEGIN OLDCODE`) at the beginning of the commented section.

  Similarly, you may want to clearly delimit the new code you write in the file, to keep track of your changes. Such marking should help even if you are using a `CVS`.

- You may have to declare **global variables** to make your code work. Although this isn not the best programming practice, it makes life much simpler with such legacy code. Be sure to use the C modifier `static` to limit the scope of your global variables to a single C file.

- To **debug** your code, you could either use

  * `fprint(stderr,...)`, or
  * `gdb`.

- You have to **comment** your code. In the event that your assignment is not completely correct, commented code is the only way we can understand your code, and give you partial credit. Also, your own code should be clearly demaracted with `BEGIN NEWCODE` and `END NEWCODE` comments.

3.1.1 Tips on using `gdb`

To use `gdb`, you have to configure and compile PostgreSQL with debugging enabled, as below:

```
./configure --prefix=PREFIX --enable-debug --enable-depend
```

This creates much bigger compiled binaries, though. For this assignment, you will probably not need to resort to `gdb`.
If you do decide to use `gdb`, you should run it as below:

```
$gdb postgres
(gdb) run -B 20 -D ~/data/ -d 1 -s mydatabase < ScanQueries.sql
```

4 Evaluation of the performance

Finally, you will now compare the performance of the `LRU`, `LFU`, and `MRU` replacement policies. To accomplish this, you will have to run the queries (from your query files) with varying buffer cache sizes (as discussed in Section 2.3). The PostgreSQL binary should report *buffer hit rates* for each query in the query files. For each run, you should take the average over all queries in a file.

1. You should **produce** 2 graphs, as described below:

   - The first graph contains the buffer hit rates for all three algorithms, applied to the queries on `ScanQueries.sql`, using varying buffer sizes.
   - The second graph contains the buffer hit rates for all three algorithms, applied to the queries on `IndexScanQueries.sql`, using varying buffer sizes.

   Remember to clearly mark the axes on the graphs and state clearly which parameters are being kept constant and what their values are. *Submissions with graphs which are not clear and precise will be penalized*

2. **Analyze/Interpret** any "interesting" trends that you notice in the behaviour of the `LRU`, `LFU`, and `MRU` replacement policies. You have to explain when and why one policy is better than the other. Note that any trend you claim should be clearly visible in the graphs. If you notice anything else interesting in your results (say the behaviour over some type of queries), then add this to your report. *Please be concise!*

3. Try to **estimate** the size of your table `Data` in number of blocks, based on the buffer cache hit rates, with increasing size of the table. (*This should be easy!*)

5 Submission instructions

1. You have to submit **electronically** the following files:

   - Your `report`(`pdf` ONLY):

     – it should contain the graphs, your interpretation of the results, and any other interesting trends that you observed. **No** additional documentation is necessary.

   - A `README` file:

     – it should contain your name, student number, what works and what doesn't work in your assignment.

   - `ScanQueries.sql`
   - `IndexScanQueries.sql`
   - The 4 C code files for `LFU`. Rename the files
     `freelist_lfu.c, bufmgr_lfu.c, buf_init_lfu.c, buf_internals_lfu.h`.

     You will have to submit all 4 files even if you didn't have to modify all of them.
   - The 4 C code files for `MRU`. Rename the files
     `freelist_mru.c, bufmgr_mru.c, buf_init_mru.c, buf_internals_mru.h`.

     As with LFU, you will have to submit all 4 files even if you didn't have to modify all of them.
   - If you had to modify any other files, please include them as well. Note that the assignment can and should be solved by modifying just the 4 files mentioned.

Good luck!