Assignment 2

Relational Operators and Memory Buffer Management

Last updated: **Tuesday 28th March 12:04pm**

Most recent changes are shown in red ... older changes are shown in brown.

Recent Updates

- **[28th Mar]** Fix the bug of writing unexpected values in `db.c`. The bug would not affect you, and the update is just for rigor.

Aims

This assignment aims to give you

- an understanding of how data is organized inside a DBMS
- practice in implementing simple relational operators
- practice in implementing the memory buffer

The goal is to implement two relational operators, selection and join. Given the memory buffer slots and the page size, you need to implement your own memory buffer to read/write data files from hard drive.

Summary

Deadline	Friday 14 April, 9pm
Pre-requisites:	Buffer Pools, File Management, and Relational Operations
Late Penalty:	5% of the max assessment mark per-day reduction, for up to 5 days
Marks:	This assignment contributes **20 marks** toward your total mark for this course.
Submission:	Moodle > Assignment > Assignment 2

Make sure that you read this assignment specification **carefully and completely** before starting work on the assignment. Questions which indicate that you haven't done this will simply get the response "Please read the spec". We will provide an extension for all students if there is a major code update in the last three days. **Let us know if there is any error in the assignment specification or in the source code.**

Introduction

The course is DBMS implementation. It is time to implement a database. The database is somehow a software or a system to retrieve data from files in hard drive. Users communicate with the database via SQL, and SQL is intepreted into the combination of several relational operators. In this assignment, you are required to implement two typical relational operators, **select** and **join**. To implement the opeartors, you mush read data from hard drive and **manage the memory buffer** given that there is a limited number of buffer slots. Related lecture notes can be found as follows. Undoubtedly, a real model database involves a great amount of features and techniques. We simulate a mini database for this assignment. To simplify the problem, we make the folowing assumptions.

- The server contains only one database.
- The database never updates.
- All attributes are integers, i.e., the size of each tuple is fixed.
- Each table takes one file, i.e., there is no overflow files.
- The select condition is always the equality test for a single attribute.

Get Ready

You do not need the PostgreSQL server for this assignment. All tasks are about C programming. That means you may do the assignment in any Linux environemnt (windows is not supported since linux system functions are used). Remember to test your code on `vxdb` server before submission since we will test your code there. The following process assumes that you are on the `vxdb` server. You may change some paths according your local environment. You should start by getting the template codes.

```
... assume you are on the vxdb server ...
$ cd /localstorage/$USER
$ unzip /web/cs9315/23T1/assignment/2/ass2.zip
```

You should see a new directory called `ass2` and the following files inside.

```
ass2/
|--- README.md          // log for recent updates
|--- db.c               // global database information
|--- db.h               // definitions for all data types
|--- main.c             // main entry
|--- run.sh             // script to run the code
|--- ro.c               // relational operators
|--- ro.h               // definitions for ro.c
|--- Makefile           // compile rules
|--- test1/             // directory containing testing files
     |--- data_1.txt    // testing data
     |--- query_1.txt   // testing queries
     |--- expected_log_1.txt // expected results
```

Next, you can complile and run the code.

```
$ cd /localstorage/$USER/ass2
$ make
... should no error here ...
$ ./run.sh
... execture the program and examine the output ...
```

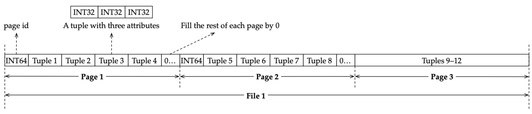
After running the code, a log file called `log.txt` will be created under the `test1/` directory.

Code structure

A great amount of comments are provided in each file. **Read all codes** to fully understand the data structure. You can start from reading `db.h` and then `main.c`. For instance, `db.h` contains a set of important data types to represent the database, the table, and the global configuration. Certain code snippets will give you inspirations to complete the task. We briefly describe what `main.c` does below.

1. load several input arguments.
2. `init_db()` reads an input `.txt` data file and produce all table files in a certain folder.
3. `init()` is an interface for your implementation.
4. `run()` loads and inteprets queries from a `.txt` query file and invoke your implementations of `sel()` and `join()` to process queries. Query results will be written into a log file.
5. `release()` is an interface for your implementation to release your data structure used for query processing.

An example of table file structure. The following figure gives an example of the file structure for each table stored in hard drive. Each table file is named as the table's oid, which is exactly same as PostgreSQL. The oid of a table can be identified from `Table.tupcons` (details). Since we are dealing with fixed-size tuples, we do not need to record the meta information in each table. For each page, we first add an INT64 to represent the page id. Then, we sequentially add each tuples until the sapce is not enough to hold a tuple. We add 0 to the end and get the whole page. The example shows a table file for three attributes. You can identify the details by reading the implementation of `init_db()` in `db.c`.



Find more details by reading the code. All following content in this specification assumes that you have already read the code and at least understood the role of each file.

Tasks

In this assignment, you need to implement two functions, `sel()` and `join()`, which represent select and join relational operators, respectively. You are also free to add any additional functions and files. In your implementation, you are given the page size (`Conf.page_size`), the number of available memory buffer slots (`Conf.buf_slots`), and the maximum allowed number of opend files (`Conf.file_limit`) at the same time.

To support the two operators, you need to write your own code for memory buffer management based on the input configuration, which means the number of data pages cannot exceed the number of buffer slots. You are only required to implement **the clock sweep policy** for memory buffer replacement, which is also used by Postgresql. **To verified your process of buffer memory for marking, you need to invoke `log_read_page()` every time you read a page from the hard drive and invoke `log_release_page()` every time you release a page from the hard drive.** The input argument for both functions are the page id. You can check what we do in these functions in `db.c`.

As introduced in lectures, opening a file is relatively costly. To improve the performance, we can maintain several opened file pointers. When reading data from a file, you can directly used the file pointer if the file has already been opened and maintained. Given the `Conf.file_limit`, you need to close a file pointer when the number of opened files reaches the limit. You are free to use any replacement stratey for opened file pointers. **To verified your process of file pointers management, you need to invoke `log_open_file()` every time you open a file and `log_close_file()` every time you close a file.** The input argument for both functions are the table oid. You can check what we do in these functions in `db.c`. You will be reduced 3/20 marks if maintaining opened file pointers is not implemented (i.e., open a file when you need, and close the file when finish reading).

Task 1: select

The function for select operator is presented as follows. You can see it in `ro.h`. For simplicity, we only consider the equality testing for a single attribute in the condition of select. Instead of giving an attribute name, we directly provide the `idx` argument, which the offset of the attribute in the tuple. `idx` starts from 0. `cond_val` is the value for equality testing of the (`idx`+1)-th attribute value. Clearly, `table_name` represents the table name. The returned value is a `_Table` pointer. See `db.h` for details of `_Table`. Below is an example of output.

```
_Table* sel(const UINT idx, const INT cond_val, const char* table_name);

# a table example, named "T1"
# the table contains three attributes for each tuple
1 2 3
5 8 13
21 34 55
2 8 6

# the following result table is derived after running sel(1,8,"T1")
5 8 13
2 8 6
```

Task 2: join

The function for (inner) join operator is presented as follows. You can see it in `ro.h`. Similar to `sel()`, we directly provide the attribute offset of each table for joining. `table1_name` and `table2_name` are names for two tables, respectively. `idx1` and `idx2` are the offsets of two attributes in `table1_name` and `table2_name`, respectively. The returned value is a `_Table` pointer. Below is an example of output.

```
_Table* join(const UINT idx1, const char* table1_name, const UINT idx2, const char* table2_name);

# a table example, named "T1"
# the table contains three attributes for each tuple
1 2 3
5 8 13
21 34 55
5 8 6

# a table example, named "T2"
# the table contains three attributes for each tuple
1 2 3
4 5 6
7 8 9
10 11 12

# the following result table is derived after running join(0, "T1", 1, "T2")
5 8 13 4 5 6
5 8 6 4 5 6
```

Assume we aim to join two tables named `T1` and `T2`. `T1.npages` and `T2.npages` denote the number of pages of two tables, respectively. For the strategy of join, you are only require to **implement the naive nested for-loop join when the number of buffer slots is not enough to hold two tables**, i.e., `Conf.buf_slots < T1.npages + T2.npages`. The number of pages for a table can be calculated by `Conf.page_size` and `Table.ntuples`. If the buffer slots are enough (i.e., `Conf.buf_slots >= T1.npages + T2.npages`), you can load all data in memory and perform an in-memory join. In this case, you need to implement **either sort-merge join or hash join**.

Submission & Evaluation

In this assignment, you are required to complete `ro.c` and `ro.h`. You may add some additional `.c` and `.h` files, and if that happens, you definitely need to update `Makefile` to make your new files work. Therefore, you need to submit at least three files, `ro.c`, `ro.h`, and `Makefile`. You need to submit all files on **Moodle**. You do not need to upload all green files marked in the file structure since we will add them to your code in testing. That means you are free to change them for developing and debugging, but all of them will be overwritten even you submit them. Specifically, below is the process to test your code.

1. download your solution.
2. copy `db.h`, `db.c` and `main.c` to the folder (the same ones as you can see in `ass2/`). Overwrite files with the same name.
3. execute `make` to compile the code.
4. copy `run.sh` and `test/` to the folder, which provide different and much more test cases.
5. execute `./run.sh` to load data and process queries.
6. compared the output log file with the expected result.

Have Fun~