387-Spring 2022- Lab5: Implementing DB internals with ToyDB

Note: You must do this assignment in pairs as you did Lab 4.

Problem Statement

ToyDB is a rudimentary relational database, which functions similar to a database, has similar properties and use cases with a subset of functionalities of a database, a small scale database. Think of a toy replica of a remote car, which is similar to a car, has similar parts, and is of miniature size and complexity.

You are given the code for toyDB with some missing parts that you have to fill in. This assignment is split into three tasks:

Task #1: Building a record layer **[marks : 25]**

First, you need to fill in the missing code in the ***dblayer*** directory. It is a record or tuple layer on top of a physical layer library (***pflayer***, that is already present). The physical layer library presents a paged file abstraction, where a file is logically split into pages.

You have to structure each page as a slotted-page structure. That is, the header at the top of the page must contain the following information:

- an array of pointers (offset within the page) to each record,
- the number of such records, and
- the pointer to the free space.

The actual record data is stored bottom up from the page. Tuples are addressed by a 4 byte **`rid`** (record id), where the first 2 bytes identify a page, and the other two are an offset in the slot header.
Note that this layer treats the record as a blob of bytes, and does not know about columns or fields.

For this part of the assignment, search for "UNIMPLEMENTED" in **`tbl.c`** and **`tbl.h`**, and fill in the relevant gaps.

Task #2. Loading CSV data into the db, and creating an index. [marks:25]

We will load up a table using the loadCSV API , from data contained in a CSV file. You will be supplied code in **`loaddb.c`** to do all the relevant parts; you just have to fill in the "UNIMPLEMENTED" parts.

The first line of the CSV file contains schema information; for example:

```
country:varchar,  population:int, capital:varchar
```

The data type can be one of **`varchar`**, **`int`**, **`long`**; the maximum size of each field is assumed to be less than the page size, and further all the fields in a row together fit in a page.

The rough steps are as follows:

for each row in the csv file :
split it up into fields
encode each field (according to type) into one record buffer
rid = Table_Insert(record)
AM_InsertEntry(index field, rid)

The idea is to use the table API you just built, which returns a record_id, then supply this record_id to a **B-Tree** indexer. The indexer takes *column_value* (of column on which index is built) and *record_id* and inserts it in a B tree index. **In this task the index should be built on the *population* column**. That code (with the prefix AM_, for access method) is made available to you. You simply have to read the docs *am.pdf* and *pf.pdf*.

Task #3. Testing: Retrieving the data. [marks: 50]

Fill in code in **`dumpdb.c`**.

dumpdb has two ways to retrieve data (depending on a command-line argument).

1. **`dumpdb s`** does a sequential scan, implemented using **`Table_Scan`**

   Sequential Scan:
   A **sequential scan** (also known as a full table scan) is a scan made on a database where each row of the table is read in a serial order.
   Sequential scan has to be implemented to scan every row of a table one by one in a serial order and dump results to stdout.

2. **`dumpdb i <condition> <value>`** does an index scan. Use the **AM_*** methods to do a scan of the index, and for each record id, invoke **`Table_Get`** to fetch the record.

Index Scan:

An **index scan** occurs when the database manager accesses an index to narrow the set of *qualifying rows* before accessing the base table; to order the output; or to retrieve the requested column data directly (index-only access).

Algorithm:

1. Open Index
2. Find the next entry in index
3. Fetch the row_id from the table
4. Print the row (row of a table)
5. Close Index

Steps 2-4 must be performed until you hit the end of the index.

In both cases, you have to decode the record to print it back in the same format as the csv file, so that we can compare the original CSV file with the version reconstructed from the database. There should be no difference. For the index scan implementation, ***fetch only the rows which satisfy the validity condition***. For example, in this implementation, if you want rows with population < 100000, then the below function call should fetch the rows that satisfy the condition.

index_scan(tbl, schema, indexFD, LESS_THAN_EQUAL, 100000);

Where indexFD is the file descriptor for index file when pf_openFile is invoked.

The basic conditions that you must have are as follows (listed a few of them).

1. LESS_THAN_EQUAL
2. LESS_THAN
3. GREATER_THAN
4. GREATER_THAN_EQUAL
5. EQUAL

Please refer to the **`am.h`** file to see all the conditions defined. Your code must be tested for all the conditions. You can manually change the conditions to test your code. The value, 100000 can be edited manually in the above **index_scan** function call.

Note(for testing the code): This process can be automated from the command line too. You need to change the method of execution from ***dumpdb i*** to ***dumpdb i condition value.*** Where you can provide the **condition** and **value** through command line input. Example: ***dumbdp i GREATER_THAN 100000*** **[ONLY FOR TESTING].** *The final submission should be executable with just 'dumpdb i' or 'dumpdb s'.*

With reference to index_scan(tbl, schema, indexFD, LESS_THAN_EQUAL, 100000);
tbl: table should be defined in tbl.h (as a part of task 1)
schema: set as default value in main() in dumpdb.c

Miscellaneous details:

1. Familiarize yourself with the am.ps and pf.ps docs on the parts that are already built. You don't need to understand all the internals though.

2. Invoke **`make`** in each of the **pflayer** and **amlayer** directories before building the **dblayer**. And finally invoke make in **dblayer** for testing the tasks.

3. Testing Task 1 - Task 1 can be tested through Task2 and Task3.

4. Testing Task 2 - Run **`"loaddb"`** . For testing, in loadCSV() print the record_id for each CSV row.

5. Testing Task 3 - Run "**`dumpdb i <condition> <value>`**" for index scan and **`"dumbdb s"`** for sequential scan. To test the working of index scan, you can verify with the condition specified.

   index_scan(tbl, schema, indexFD, LESS_THAN_EQUAL, 100000);

Calling the above function should print the rows with population less than or equal to 100000.

To test the working of sequential scan, you can verify if the rows are printed sequentially.

Submission format:

Submission Mode/ Portal - **MOODLE**

File format - **tarball (.tar)** Naming convention - **RollnumberI_Rollnumber2.tar**

Submit all the files that are provided to you by making your modifications to the required files in the 3 tasks listed above. Along with the files, submit a **README** file.

README guidelines:

Please include the following information in your Readme file.

- Names and Roll numbers
- References
- Individual contribution
- Small writeup of the execution

Other guidelines:

- Comment the code wherever you have made additions.
- Follow a proper naming convention for the variables and objects
- Attach screenshots of your execution in a separate folder named **screenshots**
- All the extra files/ folders and Readme should be inside the same **tar file** you submit.

Grading Rubric:

1. Task 1 - 25
2. Task 2 - 25
3. Task 3 - 50