CSCE-608 Database Systems

Fall 2010

Instructor: Dr. Jianer Chen
Office: HRBB 315C
Phone: 845-4259
Email: chen@cs.tamu.edu
Office Hours: T&Th 8:30am-9:30am

Teaching Assistant: Mu-Fen Hsieh
Office: HRBB 328A
Phone: 845-5457
Email: mufen@cse.tamu.edu
Office Hours: M&W 1:00pm-2:30pm (or by appointment)

COURSE PROJECT

(Due December 10, 11:59pm)

Remark. This is a project by teams of no more than two students.

This course project is to design and implement a simple SQL (called Tiny-SQL) interpreter.

The grammar for Tiny-SQL is given in Appendix 1. Your interpreter should accept SQL queries that are valid in terms of the grammar of Tiny-SQL, execute the queries, and output the results of the execution.

Your interpreter should include the following components:

- A parser: the parser accepts the input Tiny-SQL query and converts it into a parse tree. You can develop the parser using available softwares such as LEX and YACC, or write your own procedures (which should be feasible because Tiny-SQL has a very simple grammer).
- A logical query plan generator: the logical query plan generator converts a parse tree into a logical query plan. This phase also includes any possible logical query plan optimization. In particular, you should implement the ideas of pushing selections down and placing a smaller table on the left for binary operations.
- A physical query plan generator: this generator converts an optimized logical query plan into an executable physical query plan. This phase also includes any possible physical query plan optimization;
- A set of subroutines that implement a variety of data query operations necessary for execution of queries in Tiny-SQL.

For illustration of the effects of accessing disks and memory, a library StorageManager is provided for physical execution. The library simulates a disk and a fictitious memory in the real memory. Thus, the intepreter built upon the library handles data only in real memory but not on a real disk. However, the disk latency and relatively small memory is simulated in the library to show the need of query plans and optimization. The library StorageManager provides functions of evaluating the performance of the designed interpreter. A description of the library is given in Appendix 2.

Additional requirments are in the following.

- Your Tiny-SQL interpreter is recommended to include an interactive interface which accepts one Tiny-SQL statement at a line. In addition, the interpreter should be able to read a Tiny-SQL script file, each statement in a line, and be able to output the results to a file as well. Your interpreter will be tested on Tiny-SQL scripts.

- Please make sure you follow the TinySQL grammar. For examples, be careful to allow space between words, and do not allow a semicolon at the end of a statement.

- You have to implement one-pass and two-pass algorithms for the physical query plan. Possible query plan optimizations include operator push-down and join optimization.

- Design experiments to show the correcteness of Tiny-SQL execution.

- Design experiments to show running times and number of disk I/Os required by database operations using the one thousand data tuples from the project 1. Collect several measurements of running time versus data size and disk I/Os versus data size.

Write a report containing at least the following four parts:

- A User's Guide on how to use your interpreter.

- A description of software architecture or program flow. An explanation of each major part in your project, including data structures and algorithms. Please do not repeat the details of algorithms from the textbook.

- Experiments and results demonstrating the sample executions of your interpreter, and the performance of your interpreter.

- Discussion on any optimization techniques implemented in your project and their effects.

You will submit your source code, a README, and a report via CSNET. You are also expected to give a demonstration on your project in a 15-minute session.

The library StorageManager and sample Tiny-SQL statements are available for download on the TA's webpage at http://students.cse.tamu.edu/mufen/csce608/. The testing data will be available before project demonstration.

Appendix 1 Tiny-SQL Grammar

```
letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r
   | s | t | u | v | w | x | y | z
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
integer ::= digit+
comp-op ::= < | > | =
table-name ::= letter[digit | letter]*
attribute-name ::= letter[digit | letter]*
column-name ::= [table-name.]attribute-name
literal ::= "whatever-except-(")-and-(")"

statement ::= create-table-statement | drop-table-statement | select-statement
   | delete-statement | insert-statement

create-table-statement ::= CREATE TABLE table-name(attribute-type-list)

data-type ::= INT | STR20
attribute-type-list ::= attribute-name data-type
   | attribute-name data-type, attribute-type-list

drop-table-statement ::= DROP TABLE table-name

select-statement ::= SELECT [DISTINCT] select-list
                     FROM table-list
                     [WHERE search-condition]
                     [ORDER BY colume-name]

select-list ::= * | select-sublist
select-sublist ::= column-name | column-name, select-sublist
table-list ::= table-name | table-name, table-list

delete-statement ::= DELETE FROM table-name [WHERE search-condition]

insert-statement ::= INSERT INTO table-name(attribute-list) insert-tuples

insert-tuples ::= VALUES (value-list) | select-statement

attribute-list ::= attribute-name | attribute-name, attribute-list
value ::= literal | integer | NULL
value-list ::= value | value, value-list
```

```
search-condition ::= boolean-term | boolean-term OR search-condition
boolean-term ::= boolean-factor | boolean-factor AND boolean-term
boolean-factor ::= [NOT] boolean-primary
boolean-primary ::= comparison-predicate |"[" search-condition "]"
comparison-predicate ::= expression comp-op expression
expression ::= term | term + expression | term - expression
term ::= factor | factor * term | factor / term
factor ::= column-name | literal | integer | ( expression )
```

Appendix 2. Description of the Library

The StorageManager library simulates disk and memory storage in "the memory" of your computer. You will build a single-user Tiny-SQL interpreter upon the StorageManager, storing data in the simulated disk, and get the data from the simulated disk to the simulated memory for further processing. Because the storage is simulated in the memory, everytime your simulated database will start running with empty storage. When the user inputs SQL statements, such as doing insertion, deletion, updating, and querying, the interpreter should access simulated disk and simulated memory. When the database system terminates, the data in the simulated storage are lost.

The data structure of StorageManager is summerized below in a bottom-up fasion:

- Field.h: A field type can be an integer (INT) or a string ($STR20$).
- Class Tuple: A tuple equals a record/row in a relation/table. A tuple contains at most $MAX\_NUM\_OF\_FIELDS\_IN\_RELATION = 8$ fields. Each field in a tuple has offset 0,1,2,.... respectively. The order is defined in the schema. You can access a field by its offset or its field name.
- Class Block: A disk or memory block contains a number of records/tuples that belong to the same relation. A tuple cannot be splitted and stored in more than one blocks. Each block is defined to hold as most $FIELDS\_PER\_BLOCK = 8$ fields. The max number of tuples held in a block can be calculated from the size of a tuple, which is the number of fields in a tuple.

  The max number of tuples held in a block = $FIELDS\_PER\_BLOCK/num\_of\_fields\_in\_tuple$

  You can also get the number by calling Schema::getTuplesPerBlock().
- Class Relation: Each relation is assumed to be stored in consecutive disk blocks on a single track of the disk (That is, in clustered way). The disk blocks on the track are numbered by 0,1,2,.... The tuples in a relation cannot be read directly. You have to copy disk blocks of the relation to memory blocks before accessing the tuples inside the blocks. To delete tuples in the disk blocks, first copy the blocks to the memory, invalidate the selected tuples inside the memory blocks and left holes in the original positions, and finally copy the modified blocks back to the relation. Be sure to deal with the holes when doing every SQL operation. You can decide whether to remove trailing holes from a relation. The Relation class can be used to create a new Tuple.
- Class Schema: A schema specifies what a tuple of a partiular relation contains, including field names, and field types in a defined order. The field names and types are given offsets according to the defined order. Every schema specifies at most total $MAX\_NUM\_OF\_FIELDS\_IN\_RELATION = 8$ fields. The size of a tuple is the total number of fields specified in the schema. The tuple size will affect the number of tuples which can be held in one disk block or memory block.
- Class SchemaManager: A schema manager stores relations and schemas, and maps a relation name to a relation and the corresponding schema. You will always create a relation through

the schema manager by specifying a relation name and a schema. You will also get access to relations from SchemaManager.

- Class Disk: Simplified assumptions are made for disks. A disk contains many tracks. We assume each relation reside on a single track of blocks on disk. Everytime to read or write blocks of a relation takes time below:

  $avg\_seek\_time + avg\_rotation\_latency + avg\_transfer\_time\_per\_block * num\_of\_consecutive\_blocks$

  The number of disk I/Os is calculated by the number of blocks read or written.

- Class MainMemory: The simulated memory holds $NUM\_OF\_BLOCKS\_IN\_MEMORY$ blocks numbered by 0,1,2,.... When testing the correctness of the interpreter, $NUM\_OF\_BLOCKS\_IN\_MEMORY$ will be set to 10. When measuring the performance of the interpreter using one thousand 5-8 field tuples, $NUM\_OF\_BLOCKS\_IN\_MEMORY$ will be set to 300. You can get total number of blocks in the memory by calling MainMemory::getMemorySize().

  Before accessing data of a relation, you have to copy the disk blocks of a relation to the simulated main memory. Then, access the tuples in the simulated main memory. Or in the other direction, you will copy the memory blocks to disk blocks of a relation when writing data to the relation. Because the size of memory is limited, you have to do the database operations wisely. We assume there is no latency in accessing memory.

Below is a short description of how to use the library.

- At the beginning of your program, you have to create a MainMemory, a Disk, and a SchemaManager. The three objects will be used throughout the whole program.

- You have to handle the disk block addresses and memory block addresses by yourselves. The library does not decide for you where to store the data. You should always start with creating a Schema object, creating a Relation from the SchemaManager by specifying a name, say "Student", and the Schema, and getting a pointer to the created Relation. Then, create a Tuple of the Relation through the Relation class. Get a pointer to an empty memory Block, say block 7, from the MainMemory. Store the created Tuple by "appending" it to empty memory Block 7. Finally copy the memory block to the disk block of the created Relation, say, copying memory block 7 to the disk block 0 of the Relation "Student".

- When you need to browse the data of a Relation, copy disk blocks of the Relation to memory blocks, and access the tuples from the Blocks of the MainMemory. There are two ways to access the tuples. You can get a pointer to a specific Block of the MainMemory, and access the Tuples inside the Block. Otherwise, you can also access directly the Tuples stored in consecutive memory blocks by calling the functions of the MainMemory. The Tuples will be returned in a vector. It is convenient to sort or build a heap on the vector of Tuples.

- The data of each relation are assumed to be stored in a dense/clustered/contiguous way in the disk. Each relation has data stored in disk block 0, 1, 2,....,and there is no size limit.

A way to maintain dense disk blocks is: every time before you "insert a tuple", you should copy the last disk block of the relation, say block R, to the memory block M. Append the inserting tuple to the memory block M if the block M is not full. Then, copy the memory block M back to the disk block R. However, if the memory block M is full, which says the last disk block R is full, you have to insert the tuple to the next disk block R+1. Thus instead, get an empty memory block M', insert the tuple to beginning of the memory block M', and copy that memory block M' to the disk block R+1.

- When deleting a Tuple of Block R from the relation, the simplest way is to copy the Block R to memory, "null" the Tuple, and rewrite the modified memory block to the same place, Block R, in the relation. In this way, there might be holes in the Block and in the Relation, so be sure to consider the holes when doing every SQL operation. You certainly can have other way of managing tuple deletion. Bear in mind that reading and writing disk/relation blocks take time.

  You are recommended to read the usage of each class in .h files. Then, trace the test program TestStorageManager.cpp, which demonstrates how to use the library. You don't need to read the implementation of the library in StorageManager.cpp. If you have questions about how a function is implemented and used, please contact TA.