COURSE PROJECT #2

(Due December 8, 2015)

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.
- 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.
- 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. The subroutines should use the released library StorageManager, which simulates computer disks and memory.

The StorageManager library is provided to support physical execution of the Tiny-SQL interpreter. The library simulates a fictitious disk containing 100 tracks of unlimited length and a small fictitious (main) memory of 10 data blocks. The StorageManager is configured to simulate the speed of 50M memory and a 300M relation on a Megatron 747 disk when the memory capacity is 10 blocks and the relation holds 60 tuples. With the hardware limit, the Tiny-SQL interpreter must be implemented wisely using query plans and algorithms. Otherwise, it will not be possible to handle small data such as two relations of 60 tuples or six relations of only 5 tuples.

Further descriptions of the library are given in Appendix 2.

Specific instructions and requirements of the project are as follows.

- Interface: Your Tiny-SQL interpreter should have an interface. Single-user text-based interface is sufficient. The interface accepts one Tiny-SQL statement at a line. In addition, the interface should be able to read a file containing many Tiny-SQL statements, one statement per line, and be able to output the query results to a file.

- Parser: You can develop the parser either by writing your own procedures (which should be feasible because Tiny-SQL has a very simple grammar) or by using a parser generator such as LEX, YACC, and JavaCC, if you are familiar with them. We would recommend that you write your own parser, in particular for students who do not have extensive experience in compiler constructions.

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

- Logical query plan generator: You will need a tree data structure. *You should at least optimize the selection operations in the tree.*

- Physical query plan generator: *You should at least optimize the join operations.*

- Implementation of physical operators: You should implement the one-pass algorithms for projection, selection, product, join, duplicate elimination, and sorting as well as the two-pass algorithms for join, duplicate elimination, and sorting.

  Hints: (1) Based on the StorageManager, you can use at most 10 memory blocks; (2) Among the join operations, cross-join is easier to implement but will not work for large relations; and (3) To handle WHERE conditions, it is suggested that you do not further decompose the condition into a subtree. Instead, keep the entire condition as a single node in your parse tree.

Read Appendix 3 on how to start your project.

Testing will be done in two ways:

- A set of data and queries are provided to test the correctness of your interpreter. The data include 11 tables and as much as 60 tuples in one table. A file is provided for you to record the number of disk I/O's used by the interpreter. You do not have to worry about Tiny-SQL statements outside the test set.

- Use your 200-tuple data from Project #1. Design experiments to show the running time and the number of disk I/O's required to execute sample queries. Collect several measurements of running time versus data size, and disk I/O's versus data size.

Any extra optimization techniques for Tiny-SQL statements implemented in your project will be considered as bonus points.

The library StorageManager and sample Tiny-SQL statements are available on the TA's web page at `http://cuiyi.org/teaching/CSCE608/`.

Write a report containing at least the following components:

- A description of the software architecture or the 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 all the queries your interpreter is capable to execute, 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 note of how to install your software, and the report via CSNET. Also submit a hard copy of the report.

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. Additional Descriptions of the StorageManager Library

The StorageManager library supports physical execution of Tiny-SQL interpreter. The library simulates a fictitious disk containing 100 tracks of unlimited length and a small fictitious (main) memory containing 10 data blocks. To simplify the database system, every relation is stored separately on one disk track. Each of the disk tracks and the memory is partitioned into blocks of the same size. The blocks are numbered in sequence, and the data are transferred between the disk and the memory block by block.

The StorageManager is configured to simulate the speed of 50M memory and a 300M relation on a Megatron 747 disk where the memory capacity is 10 blocks, and the relation holds 60 tuples. To achieve the effect, each block is configured to hold only 8 data fields. The disk latency is set up according to the Megatron 747 disk but 320 times slower for transfer time of one block. The performance of a Tiny-SQL interpreter varies when implemented in different ways. Additional library functions are provided to time the performance of your Tiny-SQL interpreter.

In addition to hardware simulation, the StorageManager supports creation and deletion of a relation with a specified schema. Thus, you do not need to worry about how to store a relation on the fictitious disk. When executing queries, the Tiny-SQL interpreter only needs to specify which relation block to transfer to which memory block or the opposite direction. However, in order to locate a tuple, you must know for each particular relation, how many tuples are stored in one relation block. Every relation/memory block stores a fix number of 8 fields. Because different relation has a different schema and a different number of fields, the number of tuples a block can hold varies from one relation to another.

Every time your Tiny-SQL interpreter will be started with an empty disk and an empty memory, because the StorageManager library simulates the fictitious disk and memory only in the program memory. When the interpreter terminates, the data in the simulated storage are lost.

The data structure of StorageManager is summarized below in a bottom-up fashion:

- Field.h: A field type can be an integer *INT* or a string *STR20*.
- Class Tuple: A tuple (i.e. a record/row in a relation/table) 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/relation or memory block contains a number of records/tuples that belong to the same relation. A tuple cannot be split and stored in more than one blocks. Each block is defined to hold at most $FIELDS\_PER\_BLOCK = 8$ fields, but different number of tuples. The maximum number of tuples held in a block can be calculated from the size of a tuple, i.e. the number of fields in a tuple:

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

  You can also get the number by simply calling Schema::getTuplesPerBlock().
- Class Relation: Each relation is assumed to be stored in consecutive disk blocks on a single track of the disk (i.e., in a 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. For example, to "delete" a tuple in a relation block, you should first copy the block to the memory, invalidate the selected tuple inside the memory block, and finally copy the modified blocks back to the relation. Please NOTE that a hole is left in the block after the deletion. 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 provides functions to create a new Tuple.

- Class Schema: A schema specifies what a tuple of a particular 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 held in one relation 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 get access to relations from SchemaManager.

- Class Disk: Simplified assumptions are made for disks. A disk contains 100 tracks. We assume each relation reside on one single track of blocks on the disk. Reading or writing 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/O's is calculated by the number of blocks read or written.

- Class MainMemory: The simulated memory holds $NUM\_OF\_BLOCKS\_IN\_MEMORY = 10$ blocks numbered by 0,1,2,.... You can get total number of blocks in the memory by calling MainMemory::getMemorySize().

  When accessing data of a relation, you have to copy the disk blocks of the 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 a relation. Because the size of memory is limited, you have to implement 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.

- Create a relation: You should always start with creating a Schema object. Then create a Relation from the SchemaManager by specifying a name, say "Student", and a Schema. The SchemaManager will return a handle/pointer to the created Relation.

  NOTE: You have to handle the addresses of disk and memory blocks by yourselves. The library does not decide for you which block to store the data on a disk track or in a memory.

- Create a tuple: Create a Tuple of the Relation using the Relation pointer. It may sound weird, but you have to store the newly created Tuple to the simulated memory to complete this step. First get a pointer to an empty memory Block, say block 7, using the MainMemory object. Store the created Tuple by "appending" it to the empty memory Block 7.

- Store a tuple to a relation: Copy the memory block containing the tuple to a block of the created Relation, say, copying memory block 7 to the block 0 of the Relation "Student".

- Browse a relation: Each relation is stored in a dense/clustered/contiguous way on one disk track, i.e., the data are stored in consecutive disk blocks 0, 1, 2,. . . .,and there is no size limit.

  To browse data of a Relation, copy relation blocks to memory blocks, and access the tuples in the Blocks of the MainMemory. There are two ways to access tuples in the MainMemory:

  - You can get a pointer to a specific Block of the MainMemory, and access the Tuples inside the Block.
  - Otherwise, you can get a number of the Tuples stored in consecutive memory blocks by specifying the range of the Memory blocks. The Tuples will be returned in a vector. It is then convenient to sort or build a heap on the vector of Tuples.

- Append a tuple to a relation (and maintain clustered disk blocks): Every time before you "append a tuple", you should copy the last relation block, say block R, to the memory block M. Append the new tuple to the memory block M if the block M is not full. Next, copy the memory block M back to the relation block R. However, if the memory block M is full, which says the last relation block R is full, you have to append the tuple to the next relation 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 relation block R+1.

- Delete a tuple from a relation: Assume we want to delete a Tuple from the Block R of a relation. The simplest way is to copy the Block R to memory, "null" the Tuple, and write the modified memory block back to the relation block R. In this way, there might be holes in a Block and a Relation, so be sure to consider the holes in 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.

- Temporary relations: You will need to create temporary relations on the emulated disks to store intermediate query results. Be aware that field names from joining relations might be the same, which might cause problem for the temporary relations. You will have to take care of the problem.

You are suggested 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 do not 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 the TA.

Appendix 3. How to Start the Project

1. You should have C++/Java preliminary knowledge:
   - object-oriented programming;
   - the standard data structure vector/ArrayList; and
   - how to make heap on a vector using C++. Java users must implement heaps.
2. You should know how to define and use tree data structure.
3. Read textbook chapters (Ed. 2009):
   - Data storage: 13.2 (Disks), 13.3 (Using secondary storage effectively), 13.5-13.6 (Representing data elements)
   - Logical query: 5.1-5.2 (Logical query languages), 16 (Query compilers)
   - Query execution: 15.1-15.5
4. Once you understand the data storage, read the StorageManager description, and trace the object-oriented test program TestStorageManager.cpp. Basically every function available from the StorageManager appears in the test program.
5. Because of the simple structure of Tiny-SQL, you may not have to construct the formal parse tree, as long as you can correctly identify the necessary components in the input query. In particular, we recommend that you keep the entire condition in a WHERE clause, instead of further decomposing it into a parse subtree. The condition can be converted into an expression in the postfix form that can be easily manipulated by a stack structure.
6. Possible project time-line (You might need to adjust the order of implementation. If you fall behind, seek help soon.):
   - Week 1:
     - Read about data storage, and the StorageManager library.
     - Start your implementation in both ways, from top down and from bottom up. At the top end, write the interface and the parser. At the bottom end, implement procedures of CREATE TABLE, DROP TABLE, INSERT, DELETE and single-table SELECT *. NOTE: If you are using a parser generator, you should aim at finishing the parser in one week.
   - Week 2:
     - At the parser end, implement evaluation of WHERE clauses. Start with a single-table version, but bear in mind this need to be expanded to multiple tables.
     - At the physical layer, read about query execution. Implement product operations for multiple tables. Integrate the product procedure with the parser that handles WHERE clauses. Now your interpreter should be able to handle most of test queries for small tables.
     - Testing: When running test queries, first use 6 and 3 tuples for tables course and course2, respectively; 5, 3, and 1 tuples for tables r, s, and t, respectively; and 2, 2, 2, 1, 1, and 1 tuples for tables t1, t2, t3, t4, t5, and t6, respectively. If the results are correct, continue with larger tables. Use 60 and 16 tuples for course and course2, respectively; 41, 3, and 3 tuples for r, s, and t, respectively; and 5, 4, 4, 2, 2, and 1 tuples for t1, t2, t3, t4, t5, and t6, respectively. You should have no problem handling this amount of data in this stage.

- Week 3:
  - From top down, read about logical query and start implementing logical query tree. The tree may contain operators such as projection, selection, product, join, duplicate elimination, and sorting. You will need to modify the whole project to integrate the logical query tree into the project.
  - From bottom up, implement two-pass natural-join algorithm. Create temporary relations if necessary. Optimize the join tree.
  - Testing: Use all the tuples provided, that is, 60 and 24 tuples for course and course2; 41, 41, and 41 tuples for r, s, and t; and 5, 4, 4, 2, 2, and 1 tuples for t1, t2, t3, t4, t5, and t6.
- Week 4:
  - Implement logical query tree and its optimization.
  - Implement operations such as projection, duplicate-elimination and sorting for single table. Expand to multiple tables.
- Week 5: Project integration
- Week 6 Performance evaluation and report writing
- Week 7: Testing and bug fix