TinyDBMS (CSCE 608 Project 2)

Date: 12/03/2015
Xiaopei Xu <xvxiaopeisjtu@gmail.com> **UIN: 925000984**
Chaofan Li <li3939108@gmail.com> **UIN: 923002382**

This course project is to design and implement a simple SQL(called Tiny-SQL) interpreter. We use C++ to implement the entire project with the released library *StorageManager*, which simulates computer disk and memory.

Our project is consisted with 5 parts, which are Interface, Parser, Logical query plan generator, Physical query plan generator and Implementation of physical operators. After finishing the project, we use a set of data provided to test the correctness of our interpreter. Some tuples of data from our project #1 are also used to test the running time.

Optimizations:

1. The push-down of σ queries.
2. Natural join operations.
   There is actually no explicit natural join operation.
   All natural join operations are optimized from product operations and where conditions such as
   *SELECT * FORM course, course2 WHERE course.sid = course2.sid*
3. Join-tree optimization.

1. Compilation

Due to the large number of queries, we increased two parameters in the storage library: `MAX_NUM_CREATING_RELATIONS` in `Relation.h` and `NUM_TRACKS` in `Disk.h`. Since the library stores one relation in one track, the two parameters actually need to be the same value, or a segmentation fault occurs when more than `MAX_NUM_CREATING_RELATIONS` tables are created. So, before compilation, these two parameters need to be changed. After that, simply execute

```
$ make
```

Also, the lexical analyzer `flex` and C++ compiler `g++` are required to compile the source files.

2. Interface

The interface is simple. After compiling the interpreter, you can run it by either

```
$ ./tsql
```

which opens an interactive interpreting environment, where you can enter SQL queries and get outputs, or

```
$ ./tsql some-Tiny-SQL-input-file
```

which parses the whole file line by line and prints the query outputs.

3. Parser

The parser is generated using Flex. Considering the simple syntax of TinySQL, we don't use parser

generator such as YACC and Bison. Without using formal parser generator, we don't create formal parser tree, either. Specifically, we use the state stack of flex to construct the parser. For example, the DELETE FROM statement is parsed as such:

```
<INITIAL>{D_F}{WS}+ {
    cout <<""<< "DELETE FROM " ;
    err_out_START("DELETE FROM");
    BEGIN(DELETE_STATEMENT);
}
<DELETE_STATEMENT>{name} {
    cout << yytext << " " ;
    error_output << "delete table:\t" << yytext << endl ;
    table_name = string(yytext) ;
    BEGIN(D_S_EXPECT_WHERE);
}
<D_S_EXPECT_WHERE>{WS}*{W}{WS}* {
    cout << "WHERE" << " " ;
    yy_push_state(WHERE_SUB_STATEMENT) ;//Enter another state
}
```

{WS} matches white spaces, i.e., space and tab. The new-line character is significant. So, it doesn't match new-line character. {w} matches "WHERE", and {D_F} matches "DELETE FROM". After the interpreter matches "DELETE FROM", it enters state DELETE_STATEMENT. In that state, if it matches a table name, it then enters state D_S_EXPECT_WHERE. Then, if it matches WHERE, the interpreter enters WHERE_SUB_STATEMENT. Because the where clause is used together with both SELECT statement and DELETE-FROM statement, the interpreter may also enter WHERE_SUB_STATEMENT state from states other than state D_S_EXPECT_WHERE.

To parse the WHERE clauses, we use *shunting-yard algorithm*, which is used to generated the Reverse Polish Notation (RPN) format. In our project, we used two stacks for both final outputs and operators rather than one output queue and one operator stack in the original algorithm for generating (RPN). An expression tree is constructed during this procedure, although we don't create all the whole Abstract Syntax Tree (AST).

4. Logical query plan generator

The logical query plan generator constructs a logical query tree structure, and an expression tree structure particularly for the where clauses. The logical query tree structure is shown below:

```
enum Qtree_TYPE {
    DELTA ,//duplicate elimination operation
    PI   ,//projection operation
    SIGMA,//select operation
    PRODUCT ,//cross-join operation,
             //which can optimized to natural join operation,
             //for some cross-join operation and select operation
    TAU  ,//sorting operation
    TABLE,//table scan operation: get all tuples from specified table
    INS  //Insert operation: insert selected tuples to a table
};
class Qtree {
    public:
    int type ;
    vector<string> info;
    Qtree *left, *right, *parent;
    Qtree(enum Qtree_TYPE type, Qtree *parent);
    void print( int );
    void free() ;
    vector<Tuple> exec(bool print, string *table_name);
};
```

Each tree node has two children node, and a parent node. If it has only one child, only the left child is used. After we have logical query tree, an execution function vector<Tuple> exec(bool, string *); is used to execute the logical queries, where the physical queries are finally called. The method `Qtree::exec` is a recursive function, which traverse the tree from the root node down to the leaf tree node. Some intermediate relations are created if needed.

4.1. Logical query optimization

We also implement the push-down optimization and natural-join optimization for the logical query plan. The optimization is performed over the expression tree. Which is also shown here. The `number` field is used to store precedence if the `type` is `OPERATOR`, or the actual number if the type is `INTEGER`. The `tables` field stores all tables that are used in the expression. The method `optimize_sigma` is used to perform the push-down optimization, while the method `optimize_join` is used to perform the natural-join optimization. Both methods traverse the tree.

```
enum Qexp_TYPE{
    Qexp_ILLEGAL ,
    COLUMN  ,//leaf tree node with a column name
    LITERAL, //leaf tree node with a string
    INTEGER, //leaf tree node with an integer
    OPERATER ,//an operator tree node
    LEFT      //special node for left parenthesis: used with the operator stack
};

/* precedence  */
#define TIMES_DIVIDES 5//* /
#define PLUS_MINUS    4//+ -
#define COMPARE       3//comparing operators including >,<,and =
#define NOT_PCD       2//NOT
#define AND_PCD       1//AND
#define OR_PCD        0//OR

class Qexpression {
    public:
    enum Qexp_TYPE type ;
    int number;
    string str;
    set<string> tables ;
    Qexpression *left, *right;
    ... ...
    Qexpression* optimize_sigma(map<string, Qexpression *>* sigma_operation) ;
    Qexpression* optimize_join(vector<string> &commons,
                               map<string, bool> &joined_key) ;
    bool judge(Tuple t);
    void print(int level) ;
    void free() ;
    enum FIELD_TYPE field_type(Tuple ) ;
    private:
    union Field judge_(Tuple t) ;
};
```

The push-down optimization is very simple: if the method `optimize_sigma` encounters an OPERATOR tree node which only uses one table, then the whole subtree is pruned and stored in the `sigma_operation` argument, and a NULL pointer will be returned; if it encounters an AND OPERATOR node, the method is recursively called on its children tree node; otherwise, it returns with itself.

The natural-join optimization is performed in a similar way, except that the method will look for tree node with two tables involved and check its children nodes. If the node represents

something like `table0.field = table1.field`, then the tree node is pruned and the common field is stored in the argument `commons`.

5. Physical query plan generator

In this part, we construct the PQP. We translate the LQP to PQP by replacing the node in LQP by operation algorithms. Some operations can be done in one pass or two pass, we applied proper algorithms on it.

We optimize the join operations, and an extra optimization of join tree is also involved.

The join operations we implement are product and natural join (one pass and two pass sort-based). For join operations, we use a function to handle them:

```
vector<Tuple> physicalOP::JoinTables(vector<string> relation_names,
                                     vector<string> common_fields)
```

The input of this function are relations' names and the output are the tuples of result table. This function can do:

1. Firstly estimate the size parameter for every relation, we call a function:
   relation_data physicalOP::RelationCount(string relation_name)
   to get the V(number of different values in each field), T(tuples). This may take some time(I/Os), but will save more I/Os when we run join operations.

2. We construct a join tree for the relations using dynamic programming, the data structure of tree node is:
   ```
   class JoinNode
   {
   public:
       JoinNode *left;
       JoinNode *right;
       relation_data m;
       ...
   }
   ```
   The algorithm for constructing a join tree is taught in class, we don't repeat it here.
   We used dynamic programming to get the size and cost of each combine of relations to join. Finally, we get an optimized join tree and get the way to join with least estimate cost;

3. Follow the join tree to join these tables. We do it recursively by traversing the tree and call join for each join node.
   ```
   vector<Tuple> JoinTree(JoinNode & root, vector<string> common_fields);
   ```

   We also judge that if that if this join can be in one pass, then apply proper algorithm.

   At last we call the subroutines and get the results.

6. Implementation of physical operators

We implement physical operators needed:
Projection, selection, product, join (one pass and two pass sort-based), duplicate-elimination (one pass and two pass sort-based), sorting (one pass and two pass sort-based).

These operations are implemented in file physicalOP.h/cpp as a set of operators to be called, we

design it in singleton pattern:

```
class physicalOP{
private:
	MainMemory mem;  Disk disk;  SchemaManager   schema_manager;
	static physicalOP *physicalop;
	physicalOP():schema_manager(&mem, &disk) {
		disk.resetDiskIOs();
		disk.resetDiskTimer(); }    //singleton
public:
	static physicalOP *getInstance() {
		if (physicalop == NULL) {
			physicalop = new physicalOP();
			cout<<"Physical operator initialized!"<<endl;
			physicalop->displayMem();
		}
		return physicalop;
	}
	... ...
}
```

So when we want to use physical operators, we shall call:

```
physicalOP* p=physicalOP::getInstance();
```

Then we can call like those:

```
p>DropTable(relation_nameA);
p->CreateTable(relation_nameA,field_names,field_types);
```

Projection:

Projection is very simple and we just input the tuples and get the content in the field we need.

Selection:

We implement selection in a function:

```
vector<Tuple> singleTableSelect(string relation_name, condition con);
```

In this function, we input the relation name and the condition, then we read every block of the relation to main memory, and for each tuple judge if this tuple meet the condition, if it does, we will output it. Finally this function return a vector of Tuples for further use. (We don't store it back here)

Product:

Cross join function is:

```
vector<Tuple> Product(string relation_name1, string relation_name2);
```

The input is the relation names of each relation, and this function will judge if there are fields in these two relations with same field names. If there are same names, for example, relation A has a field named "b" while relation B has "b" field, then we will renamed it in the results as "A.b" and "B.b".

The product is implemented in one pass. This function will choose the smaller relation to get it in main memory, and for each tuple of larger relation, we cross join them and output the results. We create a new Schema (as well as a new Relation) to create new kind of Tuple for results. The algorithm is simple and has been introduced in class and textbook, so we don't repeat it detailedly.

Sorting:

We introduce the implementation of sorting before other remain operators because of that other two pass algorithm is based on sorting in our implementation. We defined some functions for sorting:

```
vector<Tuple> sortOnMemory(string field_name, int start_block,int num_blocks);
vector<Tuple> SortOnePass(string relation_name,string field_name);
vector<string> sortedSub(string relation_name,string field_name);
```

```
tupAddr getMin(string field_name,int start_block,int num_blocks);
vector<Tuple> SortTwoPass(string relation_name,string field_name);
```

Function ***sortOnMemory*** sort the tuples in memory, from start_block to start_block+num_blocks, by the order of the field_name. It also return a vector of sorted Tuples.

Function ***SortOnePass*** is one pass sorting, it can read in all the blocks of the relation to main memory and call ***sortOnMemory***.

Function ***sortedSub*** will make sorted sublists of the relation, each sublist is a relation. We read in 10 blocks once(or all blocks if number of remain blocks of original relation is less than 10), sorted them, create a relation and write blocks back to these sublists. Then we return a list(vector) of sublists' names for further use.

Function ***getMin*** is a function to return the location of the minimum tuple:

```
class tupAddr{//location if a tuple in mem
public:
       int block_index;
       int offset;
       ...
}
```

It is helpful when we need to apply one pass algorithms on sublists.

Function ***SortTwoPass*** firstly use the function ***sortedSub*** to make sublists, then from each of them read one block to memory and output the minimum. It act as the algorithm introduced in text book.

Duplicate-elimination:
We designed some functions for duplicate-elimination:

```
bool tupleEqual(Tuple a,Tuple b);
vector<tupAddr*> findDupOnMemory(Tuple t,int start_block,int num_blocks);
vector<Tuple> dupOnePass(string relation_name);
vector<Tuple> dupTwoPass(string relation_name);
```

Function ***tupleEqual*** is used to judge if two tuples are same. This function compare all field in two tuples, if they are all same, return true, else return false.

Function ***findDupOnMemory*** is to find same tuples as tuple t in Memory (from start_block to start_block+num_blocks). And it will return these tuples address(block number and offset).

Function ***dupOnePass*** just read all blocks of relation to memory, then for each tuple call ***findDupOnMemory*** to find duplicate and null these tuple except one. Then output the tuples without duplicates.

Function ***dupTwoPass*** is two pass duplicate-elimination and it is more complex than one pass. We won't repeat the algorithm in detail in this report either. We also call the function ***sortedSub*** to make sorted sublists, and then eliminate the duplicate of minimum. This function also output the a list(vector) of tuples without duplicates.

Join(natural join):
Functions about join are:
vector<Tuple> JoinOnePass(string relation_name1, string relation_name2, vector<string>

common_fields); //natural join, one pass

Tuple JoinOneTuple(string relation_name1, string relation_name2, Tuple t1, Tuple t2, vector<string> common_fields); //return a joined tuple if can join, or an invalid tuple

vector<Tuple> JoinTwoPass(string relation_name1, string relation_name2, vector<string> common_fields); //natural join, two pass

Function ***JoinOneTuple*** is to join two tuples, it will return a joined tuple if these two can be joined, otherwise it will return an invalid tuple.

Function ***JoinOnePass*** firstly read all blocks of smaller relation to memory, and then for each tuple in larger relation, call ***JoinOneTuple*** for tuple of larger relation and every tuple of smaller relation. Output the result tuple if these two tuples can be joined.

Function JoinTwoPass will call ***sortedSub*** for each relation and make 2 sets of sublists, and then we apply the two pass algorithms for natural join. Get minimum tuples of each relation, and join them. The output is a list(vector of) joined tuples.

Also, common_fields is needed to decide which field is the "same" field.

Create table, drop table, insert and delete operations are also implemented in this part.

Create table:

There are two functions for creating table:

Relation * CreateTable(string relation_name, vector<string> & field_names,

vector <enum FIELD_TYPE> & field_types);

Relation * CreateTable(string relation_name, vector<Tuple> tuples);

The former one is to create a relation named relation_name, with all its field names and each field's type(INT or STR20). It will return a pointer to the new relation.

The latter one is to create a relation with tuples. If we just create a relation and insert these tuples, we must do a lot of I/Os. So we just create the relation, fill the blocks and insert them block by block.

Drop table:

void DropTable(string relation_name);

Just call

schema_manager.deleteRelation(relation_name);

to drop a table.

Insert:

Insert contains the step of create a tuple, read last block of relation, write the tuple in the block, write back the block to relation, we have two functions to handle these:

```
void appendTupleToRelation(Relation* relation_ptr,   MainMemory& mem,
                           int memory_block_index,   Tuple& tuple)
void insert(string relation_name, vector<string> field_names,
            vector<string> STR,    vector<int> INT);
```

Function ***appendTupleToRelation*** can be used when we have a tuple and we want to insert it in a relation. We read in the last block of the relation into main memory and check if it is full. If it is not full, append a tuple to this block and write it back to disk, otherwise we allocate a new block with this tuple to the relation.

Function ***insert*** is to create a new tuple and insert it in a relation. We create a tuple of the relation and set some field of this tuple of the value we provide in vectors. Then we call ***appendTupleToRelation*** to insert this tuple.

Delete:

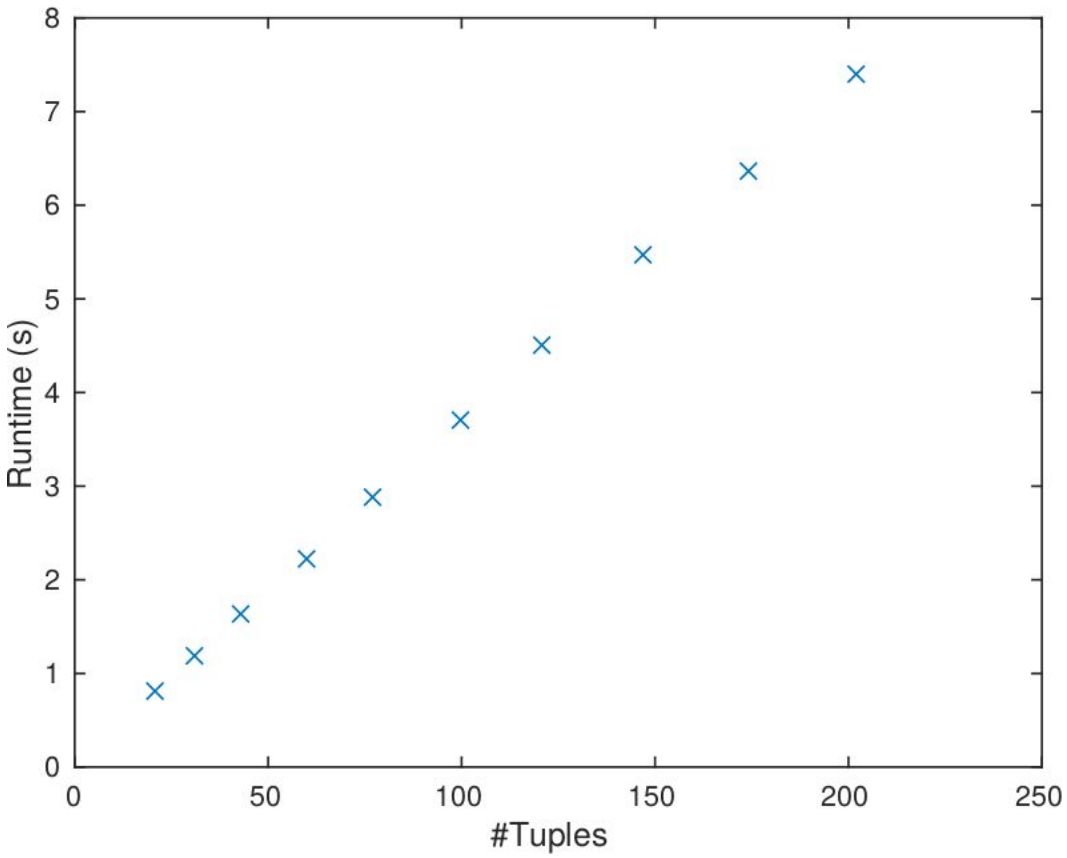
```
void physicalOP::Delete(string relation_name, condition con)
```

In ***delete*** we read in all the blocks of relation and for every tuple judge if the tuple meet the condition. Then we null the tuples meet the condition and write all remain tuples back. Empty blocks will be deleted.

In addition to operators, we have implemented some functions to display the memory or tables (for debugging and testing), and there are also some function to reset or show the I/Os.(We don't describe them in this report.)

7. Test

We perform tests over two test-cases: one provided in the TA's website (`TinySQL_linux.txt`), the other extracted from our project #1 SQL queries (`testCase2`). Both works with our program. The platform for testing is a 64-bit linux machine with flex 2.5.39, g++ 4.9 and GNU/make installed. The query outputs for testcases `TinySQL_linux.txt` and `testcase.txt` are in files: `TinySQL_linux.output` and `testcase.output` respectively. In testcase.txt we insert 202 tuples for a relation, and test the the performance of SELECT statement with varying number of tuples, which is shown in the plot. We can see that the runtime increase almost linearly with the number of tuples in the relation.



The results show that our Tiny DBMS can meet all requirements and get correct results. We can easily find that the more disk I/Os we use, the more time will be need for execution. For example, each INSERT statement cost 2 disk IOs except the first INSERT statement for a relation (the first one costs only 1 disk IO). Each regular SELECT statement without DISTINCT and ORDER BY will cost t disk IOs, where t is the number of blocks occupied by the relation. As expected, the algorithms have a huge effect on the performance. One pass algorithm will use less disk I/Os while two pass algorithm use much more. For instance, the duplicate elimination algorithm for relation *course* costs 180 disk IOs, which is exactly *3B(course)*.

The optimization also helps a lot in improving the performance of our system. Pushing selection down, using natural join to optimize cross join and optimizing the join order all reduce many I/Os. For example:

```
SELECT * FROM r, s, t WHERE r.a=t.a AND r.b=s.b AND s.c=t.c
```

we use natural join to optimize cross join and then we can optimize the join order. Finally we get a optimized physical plan and save many disk I/Os.