Angela Chan
526004664
angela.tze.chan@tamu.edu
Abilash Vallamkonda
126004453
vrabhilash@tamu.edu

CPSC-608 Database Systems Project 2
12/05/17

Language Used: C++11

The core of our project comes from the file, “dbms.cpp”, which initializes the disk and main memory, and parses the queries from a text file. As the input gets parsed, we check to see what is the first word of our query to determine the type of statement. Once we know the statement type, we will call the corresponding subroutine. Within these subroutines we are continuing to parse the query for the appropriate information we seek. Once the subroutine finishes, we would have completed reading in a whole statement. The program will continue to loop to search for the next statement until it reads a ‘0’ in the text file.

SELECT

If the first word we read in is “SELECT”, we create a “selectData” class object, which will hold the relation names as a vector of strings, the column names we wish to select as a vector of strings, and string for if there is a column name we wish to order our result by. The TinySQL grammar allows for some optional keywords. So, within this subroutine, we read the next word and put it back to the stream if it is not the optional keyword we were looking for. For eg, if the next word is DISTINCT and the read word is not, the read word gets put back into the stream. “selList” then gets called, which checks for a star. If there is a star in the Select statement, then selList returns. Otherwise, “selSublist” gets called, which calls “columnName”. ColumnName returns the string that contains the column name from the select query. Once all the column names are collected, we read the next word, “FROM”, and call “tablelist” , which calls tableName to collect the relation names. We then check if the next word is “WHERE”. If it is, then we call “createTree” to create the expression tree that stores the where clause for our query. We then check if the next words are “ORDER BY”. If they are, then we read the next word and store it to our selectData object because this is the attribute we wish to order our result by. Now, our selectData object will contain all the column names, relation names, and the order by column (if exists) from our select query.

Selecting from Single Relation

If we are selecting from a single relation, we can check if the current query is a SELECT * query by checking if the vector of column names in the selData object is empty. In this case, we use the library function “getSchema” to obtain the schema of the selected relation. Using the schema, we can call the library function “getFieldNames” to get all the column names of the relation and store it in selectData object. However, if the current select query is a standard select statement without a *, we will still get the schema of the selected relation using “getSchema”. The following program flow will apply for both types of select statements for a single relation. The next step would be to create a new relation that will contain the column names to be selected. In order to do this, we go through the column names in the selectData object and store all the field

names and field types in a vector of strings and vector of enum FIELD_TYPE, respectively. Once we have those vectors, we create a schema and create a new relation. We create a pointer to that relation and create a tuple using the library function “createTuple”. We set the field of the newly created tuple by reading each tuple of the selected relation from disk to memory and checking for its type. If there isn’t a where clause, which will be indicated by our searchTree, then we push the tuple to a vector. If there is an orderBy column in our Select query, we will will sort the vector by the Order by column name, eventually printing out the entire vector (whether or not there is an Order by). However, if we do have a where clause, we call “evalBool”, which evaluates the searchTree based on the where clause and will push the selected tuple into the result vector if the evaluation is true.

Creating the Expression Tree for the Where Clause

The where clause is read and is split into tokens using the space as a delimiter. Then, the tokens vector is recursively checked to see if it contains the operators one at a time starting with OR, followed by AND and so on. At every stage, if the operator is found then we create a node for it and split the tokens into 2 parts using the operator as the delimiter and the tokens are processed in the same way with the node representing the delimiting operator as the parent. If the operator is not found, then we just move on to check for the next one.

Evaluating the Where Clause Expression Tree

The expression tree is evaluating by using a DFS(Depth First Search) over the expression tree. evalBool() takes the expression tree and a tuple as inputs and recursively computes the operands for each operator and finally returns a boolean value to indicate whether or not the tuple satisfies the where clause.

Selecting from Multiple Relations

If we are selecting from multiple relations, we either have to execute a cross join or a natural join. We will only have to execute a natural join if there is an “=” in the Where clause and both sides of the “=” contain an attribute name. We check this by using the fact that for multiple relation queries, the attribute names are column names prefixed by relation names with a period in between. So, the check for natural join involves finding the node that contains “=” and checking its left and right children to see if both of them contain a period. For all other cases in which we select from multiple relations, we will need to perform a cross join. Once the join is completed, similar to selecting from a single relation, we store the selected tuples that satisfy all conditions into a vector of tuples. The vector containing our result then gets printed to the screen.

Cross Join

For a cross join, we read the blocks into memory 10 at a time for each relation, and store the tuples into a vector of tuples from memory. We then append the tuples read to a vector containing a vector of tuples for that corresponding relation. For every tuple in R1, we concatenate a tuple, using our “concatenate” function, from R2 and evaluate the result. If it satisfies the where condition, we collect that tuple, using our “collectTuple” function, and push it to “selTuples”, which is a vector of tuples. SelTuples is our result vector that we use to output our final result. For more than two relations, we take the result tuple after R1 and R2 has been joined, and repeat the process.

Natural Join

For a natural join, we want to use the search tree that was created from the Where clause. We check if the root of the search tree is an AND, traverse the AND nodes to check the conditions to see if there is an equality with both left and right side containing a period. If this is true, then it is a natural join of 2 relations We take the substring of left and right side of period to call sort to get the tuples of the relation in sorted order. Sort will take the relation, as provided by left substring of period and sort it by attribute name, as provided by right side of the period.

Sort will return a vector of sorted tuples. Will need to call sort for each side of equal sign so we have a sorted vector for each relation. Next, we compare the attribute names in smallest tuples in the vectors and concatenate them if they match. Finally, we evaluate the resulting tuple and output it if it satisfies the condition.

CREATE TABLE

If the first word we read in is “CREATE”, we read in the next word to see if it is “TABLE”. If it is, we create a ‘createTableData’ class object, which holds the relation name and schema of the table we want to create, and call the subroutine “createTable”. This function calls “tableName”, which returns the name of a relation. CreateTable will then call “attributeTypeList”, which calls the subroutines “attrName” and “dataType” that return the attribute name and data types for the attributes, respectively. So now the CreateTableData object holds the relation name of the table we want to create as a string, its field names as a vector of strings, and corresponding field types as a vector of enum FIELD_TYPE. This object is used to create the schema of our desired relation, which will call the library function “createRelation” to create the relation. All of our relations are stored in a hash function, “tablePtrs” (unordered_man<string, Relation*>), which holds the relation pointer using the relation’s name. At this stage, we would have read in all the words that were in the Create statement so the next word would be the start of a new statement.

DROP TABLE

If the first word of the query is “DROP”, we read in the next word and see if is “TABLE”. If it is, the “tableName” will get called, which returns the name of the relation we wish to drop. We then use the library function, “deleteRelation” to drop our table.

INSERT INTO

If the first word of the query is “INSERT”, we read in the next word and see if is “INTO”. If it is, we create an “insertData” class object, which holds are the name of our targeted relation and its schema, and call the “insertStmt” subroutine. Once inside this function, “tableName” gets called, which returns the name of the relation we wish to insert into. InsertStmt then calls “attrList”, which calls the subroutine “attrName” that returns the name of the attribute name. If attrList reads in a comma after an attribute name is returned, attrList will be called again. Once all the attribute names are collected, insertStmt will call “insertTuples”, which calls “valueList”. ValueList calls “value”, which returns the value we wish to insert into the relation. ValueList will continue to check if there are other values to collect and return. Now our insertData object holds the name of the relation we want to insert into as a string, the attribute names as a vector of strings, and the attribute values as a vector of strings. At this stage, we would have read in all the words that were in the Insert statement so the next word would be the start of a new statement.

DELETE

If the first word we read in is “DELETE”, then we create a “deleteData” class object that holds the name of the relation we want to delete from. We then call the subroutine “deleteStmt”, which will create a tree structure, using “createTree”, once a “WHERE” is read to store the condition for which tuple we wish to delete.

If there isn’t a tree generated, that means there was no WHERE clause in our Delete statement so we will call the library function “deleteBlocks” to delete all the blocks of that relation. If we do have a WHERE clause, we will read the relation blocks from disk to memory and evaluate the tuples in memory. If it satisfies the condition, then that tuple will be set to null and write it back to disk.

Some optimization techniques we implemented in our project:

a) We always evaluate the where clause on a single tuple. For a single relation query, we just pass the tuple and the searchTree into evalBool(), whereas we create the combined tuple for the relations, prepending each attribute name with the relation name. This allows us to use the same evalBool() function irrespective of the number of relations in the query.

b) The where clause is parsed into an expression tree which allows simplifies the evaluation of the where clause into a simple DFS.
c) Creating a vector of trees to include the subtrees representing conditions where one of operand is a literal. These represent the selections which can be pushed down the tree (not done in code)
d) To check whether or not we need to do a natural join, we recursively check for an equality between attribute names going to the next level of the tree whenever an AND is encountered.
e) To check for attribute name in natural join, we simply check if the terms on either side of the equality contain a period since this is indicative of the attribute name in the multiple relation case.

Results

CREATE TABLE course (sid INT, homework INT, project INT, exam INT, grade STR20)
INSERT INTO course (sid, homework, project, exam, grade) VALUES (1, 99, 100, 100, "A")
INSERT INTO course (sid, homework, project, exam, grade) VALUES (3, 100, 100, 98, "C")
INSERT INTO course (sid, homework, project, exam, grade) VALUES (3, 100, 69, 64, "C")
INSERT INTO course (sid, homework, project, exam, grade) VALUES (6, 100, 100, 65, "B")
SELECT * FROM course

```
Angelas-MacBook-Air:Project2 Skye$ ./a.out <te

1
#Disk I/Os = 0

2
Now the relation contains:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
******RELATION DUMP END******

#Disk I/Os = 1

3
Now the relation contains:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
1: 3    100     100     98      C
******RELATION DUMP END******

#Disk I/Os = 2

4
Now the relation contains:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
1: 3    100     100     98      C
2: 3    100     69      64      C
******RELATION DUMP END******

#Disk I/Os = 2

5
Now the relation contains:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
1: 3    100     100     98      C
2: 3    100     69      64      C
3: 6    100     100     65      B
******RELATION DUMP END******

#Disk I/Os = 2

6
Attributes selected:
sid     homework        project exam    grade
3       100     100     98      C

#Disk I/Os = 4
Angelas-MacBook-Air:Project2 Skye$
```

SELECT * FROM course ORDER BY exam

```
5
Now the relation contains:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
1: 3    100     100     98      C
2: 3    100     69      64      C
3: 6    100     100     65      B
******RELATION DUMP END******

#Disk I/Os = 2
```

SELECT * FROM course WHERE grade = "C" AND exam > 70 OR project > 70 AND (exam * 30 + homework * 20 + project * 50) < 60

```
6
Attributes selected:
sid     homework        project exam    grade
3       100     100     98      C

#Disk I/Os = 4
Angelas-MacBook-Air:Project2 Skye$
```

CREATE TABLE course2 (sid INT, exam INT, grade STR20)
INSERT INTO course2 (sid, exam, grade) VALUES (1, 100, "A")
INSERT INTO course2 (sid, exam, grade) VALUES (16, 25, "E")
INSERT INTO course2 (sid, exam, grade) VALUES (17, 0, "A")
SELECT course.sid, course.grade, course2.grade FROM course, course2 WHERE course.sid = course2.sid

```
7
#Disk I/Os = 0

8
Now the relation contains:
******RELATION DUMP BEGIN******
sid     exam    grade
0: 1    100     A
******RELATION DUMP END******

#Disk I/Os = 1

9
Now the relation contains:
******RELATION DUMP BEGIN******
sid     exam    grade
0: 1    100     A
16      25      E
******RELATION DUMP END******

#Disk I/Os = 2

10
Now the relation contains:
******RELATION DUMP BEGIN******
sid     exam    grade
0: 1    100     A
16      25      E
1: 17   0       A
******RELATION DUMP END******

#Disk I/Os = 2

11
1       A       A

#Disk I/Os = 6
```

SELECT * FROM course, course2 WHERE course.sid = course2.sid AND course.exam = 100 AND course2.exam = 100

```
12
1       99      100     100     A       1       100     A

#Disk I/Os = 6
```

DROP TABLE course
DROP TABLE course2

```
17
Dropped: course
#Disk I/Os = 0

18
Dropped: course2
#Disk I/Os = 0
Angelas-MacBook-Air:Project2 Skye$
```

For this next case, we created table course, inserted 60 tuples into the relation, and executed a SELECT * query. We calculated a total of 60 Disk I/Os which matches the performance of our interpreter.

```
15      100     50      90      E
15      100     99      100     E
17      100     100     100     A
2       100     100     99      B
4       100     100     97      D
5       100     100     66      A
6       100     100     65      B
7       100     50      73      C
8       50      50      62      C
9       50      50      61      D
10      50      70      70      C
11      50      50      59      D
12      0       70      58      C
13      0       50      77      C
14      50      50      56      D
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
16      0       0       0       E
17      100     100     100     A

#Disk I/Os = 60
Angelas-MacBook-Air:Project2 Skye$
```