Cullen Brown and Arijit Upadhyaya
UINs: 520003768, 423000515
CSCE 608

Project 2 Report

Implementation Details

How to install and run the software:
-Extract the contents of the zip file
-Run the command "make" to compile the code

    This should compile the following files and their header files
    Compiled Files:
    deleteUtil.cpp
    evalWhereConds.cpp
    helper.cpp
    helpSelect.cpp
    project2.cpp
    StorageManager.cpp
    dropUtil.cpp
    evalWhereConds_lastWorking.cpp
    helpInsert.cpp
    parserLib.cpp
    selectJoinUtil.cpp

-To remove object files, run "make clean"
We have our own makefile that compiles the necessary .cpp files, creates the object files and then outputs our executable named project2

To test:
Put all the queries of the TinySQL in a text file. Each query is separated by new lines.
As given in the test cases, we expect that the keywords like - CREATE, SELECT, FROM, WHERE will be in capital letters - this is a limitation of our parser.

Then to run the queries against our implementation, run the following command after running make.

```
./project2 <input filename>
```

You should see the queries being executed and DISK I/Os and time required for executing each statement(as mentioned in Piazza) after each operation.

Software architecture or the program flow:

Main:
Project2.cpp contains our main function and part of our parser. This is where the program starts. We start by creating MainMemory, Disk, and SchemaManager that are used throughout the lifecycle of the program. We then read in a line of text (our query) from an input file. We split the line on spaces and then and store it in a vector of strings. The first stored value in the vector is read to determine whether we are dealing with a creation, an insertion, a selection, a deletion, or a table drop operation. Based on the operation type, we then call a specific sub-function corresponding to that particular operation.

Creating Tables (CREATE):
When handling table creation, we call the function "Schema processCreate(vector<string> cmdStr)". This function, also described in project2.cpp, creates and returns a relation Schema by parsing through the the input vector "cmdStr". As all table creation statements take the general form "CREATE TABLE tableName (attributeOne [INT/STR20], attributeTwo [INT/STR20],…, attributeN [INT/STR20])", we pull the table name from vector index 2 and store our attribute names and attribute types in the string vector "field_names" and the FIELD_TYPE vector "field_types", respectively, from every pair of elements after index 2. These two vectors are used to create a schema, which is used alongside the table name to create a table with the Schema Manager.

According to the TinySQL implementation provided to us, there is no Disk I/O involved in this operation. We just create a Schema in the Disk for future Inserts, if any.

Inserting Tuples (INSERT):

General Case:
When the parser in our main function determines that we need to handle an insertion, we call the function "void processInsert(string line, vector<string> cmdStr, SchemaManager schema_manager, MainMemory &mem)", described in the file "helpInsert.cpp". Given that insert statements are of the type "INSERT INTO relationName (attributeA, attributeB,…, attributeN) VALUES (valueA, valueB, ... , valueN), it begins by splitting the input line by parentheses, i.e. "(" and ")". (this eliminates them from our list), after splitting our string on parenthesis we match the corresponding field name and set the values as given in the input line.

Special Case: Inserting using the SELECT statement:
Whenever the parser processes an INSERT statement, it first tries to see if there is inserting using a SELECT statement. If yes, then we set the bool variable **callFromInsert** to true and then

call the processSelect function. This will return a vector of tuples which is then used for INSERT. We loop through each of the tuples from outputTuples and insert in the Relation.

Insertion process.
For this we first need to create a Relation pointer with the table name extracted by the parser in the INSERT query. Then we create a tuple using "relation_ptr->createTuple()". Then we assign the fields of this new tuple from the Values either extracted from the INSERT statement or the tuples we got from the SELECT statement (which was inside the INSERT query). After setting the fields to the proper values(in this case we also need to take care of what type - INT or STR20 the field is), we need to write the tuple to disk. To do that, we use a modified version of the function "appendTupleToRelation" we copied from TestManager.cpp. This function copies the tuple to a block in Main Memory(MM) and then depending on whether the relation has already some blocks or is empty - it either writes the first block of the relation or just appends to the existing tuples in the relation.

Deleting Tuples (DELETE):
When the parser determines that we need to handle a deletion operation, we call the function "void processDelete(string line, vector<string> cmdStr, SchemaManager schema_manager, MainMemory &mem)" defined in deleteUtil.cpp. This function starts by splitting the input string "line" into a vector of substrings delimited by spaces and commas. It then adds all the substrings between the FROM keyword and the WHERE keyword (if there is one) of a delete command into a vector of table names (we will only have a table name between the two keywords) and adds all the substrings after the WHERE keyword (if there is one) to a vector of deletion conditions. If we did not find a WHERE condition, we merely delete all the blocks of data for our given table mentioned after the FROM keyword.

If, however, we did find a where clause, we go on to examine all the elements in our table to delete the ones that do not meet our desired WHERE conditions. We do this with a call to the function "void deleteTupleWhereClause(string table, vector<string> condVec, SchemaManager schema_manager, MainMemory &mem)" defined in the same cpp file. This function pulls all of the tuples in our chosen relation and tests them to see if they satisfy the WHERE condition (more on how our WHERE condition evaluation works is explained later); if a given tuple does meet the WHERE conditions, then the corresponding tuple is deleted.

The process:
For queries with no where condition, we call deleteBlocks() on relation_ptr at block 0. This deletes all the blocks in the relation pointer for the particular relation name defined in the DELETE command. We can always get the relation pointer from table name using schema manager. This deletes the blocks but preserves the schema for future use.

If there is a where condition, we call deleteTupleWhereClause, which loads each block from the relation to main memory and then checks each tuples that the current block contains. There is a small catch in this. We need to check all the blocks in the MM, because in some places in our disk there might be holes due to previous deletions. We then call checkTupleStatisfiesWhereJoin to see if it satisfies the WHERE condition. If yes we invalidate the tuples by calling nullTuple on that block offset. Once we are done with the whole MM block we dump it back to the relation and then bring in the next set of block. This is a one pass algorithm as for this operation, we bring as much blocks we can == NUM_OF_BLOCKS_IN_MEMORY. Once we are done with this set of blocks we bring in the next NUM_OF_BLOCKS_IN_MEMORY set of blocks until we exhaust the number of blocks in the relation we are dealing with.

The cost for delete all records is zero because we remove all the blocks in the Disk.
But if there is a where condition, we will need to go through all the blocks in the Relation, since the relation is clustered the cost of this function then is 2 Disk I/Os (a read and a write) for each tuple affected by the deletion, and since we need to check each block in the relation for tuples to potentially delete, this should cost roughly 2 B(R), where B(R) is the number of blocks in the relation. We have verified this by printing the DISK I/O needed after each Insert operation.

Dropping Tables (DROP):
When the parser in our main determines we need to handle a drop operations, we call the function “void processDrop(string line, string table, SchemaManager schema_manager, MainMemory &mem)” in dropUtil.cpp. This is a relatively simple function that attempts to call the SchemaManager deleteRelation function using the passed in string “Table”; it prints a message that reflects its success (table dropped) or failure (table could not be found). There is no cost for this operation Disk I/O wise. In this process we first need to delete all the blocks from block location 0 of the relation. Then we delete the relation by using the schema manager.
NOTE: If we do not delete all the blocks from block number 0, and instead only call the deleteRelation then in future if we create a new table with the same name.

Projection Operation (SELECT):
*When we see a SELECT keyword in the parser, we call processSelect with the command a string vector. In processSelect, we parse through the line. We have a bool to check if DISTINCT keyword exists. We then create a string vector of the tableVec. If there are multiple tables, it is needed. We then have a attribute vector - attrVec which will contain all the attributes that we need to print in the projection. Then we also create a condition Vector - condVec that is a vector of all the condition if there is a WHERE keyword. The last is a order by vector which will contain the field on which ORDER BY is needed if there is a ORDER BY keyword.

After we filled our data structures with the necessary values. We then check if there is a single table or more than one table in Table vector. If single table there is no join, otherwise there is join involved.

Projection Operations with no joins - single table involved in SELECT queries:
If there is a single table we call the printSelectNoJoinOnePass. Depending of the type of SELECT statement we have a typeSelect variable which takes in values: 0 - SELECT * FROM <table name> no where, 1 - SELECT <attribute list> FROM <table Name> no where, 2 - SELECT * FROM <table name> WHERE <condition> and 3 - SELECT * FROM <table name> WHERE <condition>

Processing:
We first see that if the number of blocks in the relation where we want to apply SELECT can fit completely in the main memory. If yes then we bring all the blocks in MM. Then we apply DISTINCT operation. Within processSelect, if the system finds the keyword DISTINCT, we set a boolean "isDistinct" to true and pass it on to our selection subfunctions. We wrote a in place One Pass DISTINCT operation for removing duplications from a relation. Since all the tuples fit in MM, we do not need to write this in a temporary schema. Now we apply the select operation i.e. we filter the tuples based on WHERE keyword if any.
After we are done with the above process, we check if there is an ORDER BY operation. If yes, then we wrote a in-place sort function that sorts a vector of tuples in-place. After we are done with all these operations then we print the corresponding tuples which is in the form of vector of Tuples - outputTuples.

For printing tuples we have two functions written. One that prints all the attributes if the statement is SELECT *, the other just prints the specific attributes as mentioned in the attrVec - the attribute vector that we get upon the preprocessing we did before in processSelect function.

DISTINCT-Duplicate Elimination-One Pass(in-memory):
If during a SELECT statement we come across the keyword "DISTINCT", we need to handle duplicate elimination. Within processSelect, if the system finds the keyword DISTINCT, we set a boolean "isDistinct" to true and pass it on to our selection subfunctions. Selection proceeds as described above until we call one of these subfunctions.

If we call printSelectNoJoinOnePass, the way distinct is handled varies based on whether or not our selection contained a WHERE condition. If the selection did contain a WHERE condition, we handle duplicate elimination by checking each tuple to see if it is a duplicate on all attributes (if we had a "SELECT *" selection) using the function "checkDupTupleAllAttr(vector<Tuple> & tuples, Tuple tuple)" or if the tuple is a duplicate on a set of specific attributes (for selections

of the form “SELECT attrA, attrB,..., attrN…”) with the function “checkDupTupleSpAttr(vector<Tuple> &tuples, Tuple tuple, vector<string> attrVec)”; each of these functions scans the whole vector “tuples” for matches to the Tuple “tuple” on the attributes specified (either all attributes or just the attributes named in attrVec); if more than one match is found (since “tuple” will be in this list, we allow for one match before we start), the matched tuple is removed from the vector. Both of these functions can be found in helper.cpp.
The we then check to see if we need to handle an ordering, and if not, output the resulting tuples.

If we called printSelectNoJoinOnePass with a query that has a WHERE clause, we process the WHERE clause (see section on where clauses) before eliminating duplicates from our set of output tuples, calling the above functions “checkDupTupleAllAttr” if we used a select * or “checkDupTupleSpAttr” if we selected on specific attributes. The results are then ordered if necessary and output.

ORDER BY - Ordering Results : One Pass(in-memory):
If during a SELECT statement we come across the keywords “ORDER BY”, in processSelect (full function information above), we store all the terms following the ORDER BY keywords into a vector of strings “orderByVec.” The function proceeds as described above, passing orderByVec on to either “printSelectNoJoinOnePass” or “processSelectJoin” based on whether or not we determine a join was necessary.

If we called printSelectNoJoinOnePass, if our select statement did not have a WHERE condition, after potentially removing duplicates from our table, we order results using the STL vector sort method, with a custom comparator that shifts lesser values to the front of the vector following a passed in string orderByAttr. The resulting sorted tuples are pushed into an output vector and sent out to be printed by the a print function. If the select statement did have a WHERE condition, we take the outputTuples array described as above and sort those tuples by a passed in string orderByAttr using the STL vector “sort” method.

DISTINCT-Duplicate Elimination - Two Pass(need a temporary schema)
If while processing the projection, we observe that the number of blocks in the relation is > MM then we need to do pass pass if there is a DISTINCT keyword. Also, we will need to do two pass ORDER BY in this case.

Logic:
If there is a DISTINCT keyword then we do a two pass DISTINCT. We have written the function - getDistinctTuples(). This function takes in a pointer to a relation pointer and then uses it to first create sorted sublists and then use the algorithm given in class to eliminate the duplicates. It then return a vector of tuples where there are no duplicates.

Now, with some luck we might have reduced the tuples so that all of them fit in MM. If yes then we do a select operation i.e. filter the tuples for the WHERE keyword if any. Then if there is an ORDER BY, we sort the tuples using one-pass(in-memory)

If still the number of tuples returned by the two pass DISTINCT operation is more than MM then we need to use multiple passes for WHERE condition. We bring as much blocks that the MM can hold. We keep track of the last index we processed for the relation so that we keep on bringing them till we exhaust the number of blocks in relation. We need to do a special case for last pass where the number of blocks can be less than then the MM size. In that case we bring the leftover blocks.

Now after applying the filter for WHERE keyword if any, the number of tuples might have decreased. If yes and the output Tuple vector size < MM, then we do a in-memory sort using the function we wrote. We use the vector sort() function for which we wrote our own comparator. If still the number of tuples is more than MM. We call the two pass sorting function which has been implemented as mentioned in the classroom slides.

The end result is an oututTuples vector which is a vector of Tuples. We pass this vector to print all the attribute or specific attribute functions according the attribute list and then this will be result of the projection which is displayed on the screen.

Evaluating WHERE condition:
For evaluating where conditions, we have implemented our own RPN evaluator for the operator AND, OR and NOT. When we see a condition vector, we call the function checkTupleStatisfiesWhere which takes in tuple, the condition vector and the schema of the tuple. This function first replaces the variables pertaining to the schema with the actual values of the tuple. Then it evaluated individual condition like exam > 100 or exam + homework > 150 or grade = “C”. This is handled by a mathematical evaluator we got from internet (specifically, Stack Overflow) and defined in the file parserLib.cpp. The evalWhereConds.cpp is our own implementation. Using the results we get for individual conditions we use a reverse polish notation stack to finally evaluate the true or false for a particular condition for a particular tuple. If the tuple is true overall for that condition we return true and then print/process further that tuple.

Projection Operations with joins - multiple tables involved in SELECT queries:
When there is join involved - it might be cross join or natural join:

In the processSelect function if we detect there are multiple tables, then we call processSelectJoin function defined in selectJoinUtil.cpp. Before calling this function in processSelect we again have the typeSelect set to 10 - SELECT * FROM <multiple tables> no where condition, 11 - SELECT <attr list> FROM <table list> no where condition, 12 - SELECT * FROM <table list> WHERE <condition - might be natural or cross>, 13 - SELECT <attr list> FROM <table list> WHERE <condition - might be natural or cross>

We have a bool variable called writeToDB which checks if more than two tables are involved in the join. If there are more than two tables, we will need to save the intermediate results in a temporary schema and use that relation to do further joins.
In this function, we have a function that checks if there is natural joins involved using the function - getJoinConditions() defined in evalWhereConds. If there are not equi-joins then the particular join is Cross Join. We explain Cross Join below:

Cross Join:
For cross join the only algorithm we have is one pass.
Initially we get the first two tables from the table vector. We call the processCrossJoinOnePass function where we pass the bigger relation pointer, smaller relation pointer , a reference to temporary relation pointer. To create the temporary relation pointer we have a function createSchemaAllAttrJoinTables that creates a new schema for joining the two tables. We use that schema to create the temporary relation pointer.

In the processCrossJoinOnePass function, we bring one block from the bigger relation to MM in the block 0 of MM. In the loop inside, we bring MM -1 block from block 1 of MM to end of MM. In this way we have all possible combinations of all the tuples of the two relations which is a cross product. After we are done we return the output tuple vector to processSelectJoin.

Here we verify if there are more tables to be joined, we write the tuples in the DB/Disk depending on the writeToDB bool. Then we process as above for further joins. We carry on like this till there are no tables to be joined.

After we are done, we print the output tuple Vectors depending on the the attribute vector. If it was * we print everything calling the printJoinedTupleAllAttr function. If specific attribute was specified then we call the function printJoinedTupleSpAttr function.

NOTE: In case of multiple table joined we need to use a new table name every time for the temporary schema. we append a number to the table name to keep it unique. Also while printing we need to trim the extra temptable name from the attribute list.

Memory wise, this should cost at least an amount on the order of the number of blocks in our small relation; we will use up to MM -1 blocks for storing the smaller relation, and one more block to pull in data from the larger relation (M >= B(small)).

Time wise, this should cost roughly the sum of all blocks in both relations worth of I/Os (B(R) + B(S)); we only read each block once, and do not store the results back to the disk, unless used as a subquery.

Evaluating WHERE conditions:

In case of cross joins for the where conditions we are evaluating using the same where evaluation we discussed above for single table. Our function is generic to handle these cases.

Natural Joins:
We have implemented all the three algorithms for Natural Join - one pass, two pass, and nested loop.
For all the three algorithms, we have used the same idea of cross join in the sense that if there are more than two tables are involved, we write the intermediate results in a temporary relation. For the temporary relations we need to create new schema, for that we use the same createSchemaAllAttrJoinTables function defined in helper.cpp.

NOTE: Whenever we have used temporary tables we have remembered to remove it at the end of the operation.

One Pass Natural Join:
This has been implemented in the function processNaturalJoinOnePass() defined in the file selectJoinUtil.cpp. This function is called if either of the relation has blocks < MM -1. In this case we bring all the blocks of the smaller relation into the MM from MM block 1 to the end. In the 0 block of MM, we keep it to bring one block of the bigger relation each time.
We join two relation if the satisfites the join condition.

We store the the results in outputTuples vector which is vector of Tuples. Then if DISTINCT keyword was there on the final result we apply one pass or two pass DISTINCT algorithm which has already been discussed above.

We then check if there was ORDER BY keyword and then order the tuples with that particular column value. Here also depending on the size of the outputTuples vector we call the one pass or two pass Sorting.

In the end we print the outputTuples vector by calling the specific attribute print function - printJoinedTupleSpAttr or the all attribute print(SELECT *) function printJoinedTupleAllAttr. While printing we also evaluate the WHERE conditions if WHERE keyword was used.

NOTE: We need to take care to trim the temporary table names appended to the attribute name. We need to do this because there cannot be duplicate column names in the implementation.

Like the one-pass cross join, before duplicate elimination or sorting, this should take at least the number of blocks in the smaller relation + 1 worth of memory (M >= B(small)), and cost the sum of the number of blocks in both relations worth of I/Os (B(R) + B(S)).

Two Pass Natural Join:
We implemented the algorithm discussed in the class for two pass natural joins. We call the function - processNaturalJoinTwoPass() function. This function is called if the condition of B(R) + B(S) <= M*M.
As discussed in class, we first sort them on the key and make sorted sublists by using the function createSortedSublistRelation function in the file helper.cpp. We check the min tuple for both the relations. We store them in a vector and then delete them. When we bring in new blocks from the sublists we verify that if the min tuples are there we keep adding them in the vector and deleting them. We maintain separate vectors for Big relation and the small relation. When there are no more match we join them. We keep this repeating until one of the relations' sublist is exhausted.

DISTINCT, ORDER BY, WHERE keywords and printing are handled same as in One Pass.

Our implementation uses one block of memory per sorted sublist in our relations, with each list being less than or equal to 10 blocks (or whatever amount the max amount of memory happens to be); it can be assumed the memory cost here will be at most the maximum allowable given memory constraints.

Additionally, like the in class algorithm, this should use at most three disk I/Os for each block in both relations (3(B(R) + B(S)); we need to read and write them each once to sort, and read one last time to perform the join.

Nested Loop Join:
This is implemented in the function processNaturalJoinGeneric() in the file selectJoinUtil.cpp. This function is called if both the relation are very big and two pass also can't be done to join them.

But we must remember, that the relations are clustered so the disk I/O are reduced to B(R), at a time. So we bring one block of the bigger block in block 0 of MM. From block 1 of MM to the end we bring in all the blocks of smaller relation and then we join them if they can be joined on the key. DISTINCT, ORDER BY and WHERE keywords have been handled in the same way as one pass or two pass.

Memory wise, we will be using at the very least two blocks of memory; one for a block in each relation. Realistically, because we only call this function when our relation is too big for either a one-pass or two-pass join, it's going to be much larger; expect the memory cost to be the full amount the system will allow in the worst case.

Since our data is clustered, we will have a time cost of the number of blocks in the larger relation divided by the size of the memory times the number of blocks in the smaller relation, plus the number of blocks in the larger relation (B(R)/M * B(S) + B(R))

Testing and Sample Cases
For testing, we used a number of sample text files which we will be including alongside our submission. Noteworthy files include "deleteTest.txt" used to test deletion operations, "insertTest.txt" used to test insertions, "dropTest.txt" to test for dropping operations, and "TA.txt" to run the test cases provided by the TA. Additional test text files not mentioned here have also been included that test a number of scenarios. Within these files, we tested using relations with tuples of varying sizes (from as little as one attribute per tuple to five or more attributes per tuple) to handle instances of varying numbers of tuples per block. We also tested on varying size relations to verify the efficacy of our one-pass and two-pass algorithms. For many of our test files, we compared behavior in our TinySQL interpreter to a MySQL instance running the same queries for consistency. Running times and disk I/Os can be found for each operation upon completion of that operation.

Some Interesting Stuff:
-We had to implement our own tuple comparison based on different attributes.
-We had to write to temporary schema in case of multiple joins.
-We wrote our own split function to split a string on multiple delimiters like on "=" or "." or "(" all at once (if, for example, your delimiter string was " .),", it would split the string one any one of those given characters).
-We wrote our own reverse polish notation and implemented a RPN stack to evaluate the where conditions.
-We wrote our own mathematical evaluation to evaluate complex operations for the WHERE clause.

-We wrote a function to extract the join condition to make it separate from the evaluations.
-We made our program modular as much as possible and wrote functions wherever code was repeated.

Optimization
Our code is largely unoptimized; we decided early on that full functionality of or TinySQL interpreter was more important than it running efficiently, so optimization took a backseat as we devoted time towards complete, albeit naive, physical operation of the DBMS. That said, provided more time, we would have implemented logic based rules to handle join tree optimization.

Before we pushing it to the background, we were working on a left deep tree implementation of a join tree, as described in Clifford Shaffer's "Data Structures and Algorithm Analysis, edition 3.2 (C++ version)". Essentially, it is a tree where each node recognizes its leftmost child and nearest neighbor to the right within its current level. We were also creating functions to calculate estimated costs for specific actions (i.e. duplicate elimination) as described in class. We ran out of time during implementation, so we left both of these out of our final implementation.

Future work:
We were working on extracting the specific conditions for a specific table, but we could not completely finish this. Implementing this will make our implementation a bit more optimized. We would also have worked to fully implement the LQP tree.

Test Results:
Selected results have been included with the submission file.