CSCE 608: Database Systems
Project #2

Done by,
Raaja Prabhu Uma Jaganathan – 824006931
Nitish Chandra - 825008967

Program Architecture:

- I used JavaCC to build the parser, made the parser to handle all the SQL statements in the TinySql grammar.
- I have a base class “Statement”, from which all the statements (SELECT / DELETE / INSERT / etc.) inherit and have the implementation for handling the statement.
- Based on the type of input statement determined by the parser, appropriate function to handle the statement is called.
- Parser ensures that all the parameters needed for the statement (eg., select statement might need select_conditions, etc.) are populated during parsing.
- I have a class called “StorageManager”, which initializes disk, memory and schema manager and provides links to each.

Select Statement:

- The statement is converted into LQP form, for execution.
- LQP form consists of Join, Select, Project, Distinct, Sort operations, each is implemented by a class. Each of the classes inherits from a base class “Table”, signifying that the output of each operation would be a table (relation).
- If the intermediate result is bigger than memory size, they are stored in Disk.
- I push the selectCondition to Join operation, which makes sure that only the tuple which would be potentially selected by the select statement are persisted. Join operation ensures to drop the tuples that wouldn’t be selected for the final output.
- I have implemented 2-pass algorithm for Sort, Join operations.
- Distinct operation first sorts the relation then eliminates the duplicates.
- To reduce number of Disk I/Os during Join, the two minimal sized tables are joined to form a table with size of the result. Then again two minimal sized tables are joined. This process is continued till all the tables are joined to form a single table. This is very similar to Huffman coding.

Insert Statement:

- For insertion, I obtain the last block of the relation and if the block has space for more tuples, I add the new tuple in the block, else I create a new block with the tuple and update the disk.
- If the insert has to happen from a “select” statement, I take the output of select statement and insert the tuples block-by-block.

Delete Statement:

- Here I read set of blocks that memory could hold, tuple-by-tuple I check if a tuple satisfies the condition, if not, I save it in the memory, when all the tuples in the memory are processed, the contents of the memory (except the last block if it isn’t full) are saved back to the disk. Then the subsequent set of blocks are fetched to memory and the process is repeated.

Condition checking a tuple:

- To represent conditions, I have a hierarchy of classes similar to the one in grammar.
- I recursively evaluate the conditions to determine if a tuple satisfies the condition (you may refer to Utils.java for more information, the implementation is pretty intuitive).
- If no search condition is mentioned in the query, each tuple considered a match.

Create Statement:

- I use schema manager's apis to create the relation with the given schema.

Drop Statement:

- I use schema manager's apis to drop the relation.

Running the jar file:

- Command to run the jar:
  - java -jar db2.jar <name_of_input_file_in_the_same_folder>
- I provide 2 jar files, db2.jar is with disk I/O delay; while if you do not want to wait long to see the results, you may run db2-withoutDelay.jar (I just made SIMULATED_DISK_LATENCY_ON = false in config.java).
- Please note that I was using System.in to read the queries, but as per the instructions, I had to change it to read from a file. Hence I made minimal changes, so you may encounter ParseException saying '<EOF> encountered', this is absolutely normal because my program would be expecting a query infinitely but the file terminates with EOF, hence the error. So you may ignore the exception but all the queries are successfully executed.

Experiments:

- My implementation is capable of handling all the valid statements as per TinySql grammar.
- With optimizations, I was able to see more than 10 times faster execution of the queries (I compared the time taken to execute each query with and without optimizations). Some queries ran forever (over an hour) without optimization; while with optimization, queries were pretty quick (within a minute or two).
- I also compared the output of my program with a SQL engine's output, and I couldn't see any mismatch.
- My Project #1 implementation involved foreign key constraints, so I couldn't directly run those queries here; hence I had to modify the queries to match TinySql grammar and they all produced the desired output.

Optimizations specific to TinySql (or not discussed in class):

- Using the merge order similar to Huffman coding is something I came up with, it would significantly improve the disk I/Os during join with a large number of tables.
- When pushing down search_condition in select statements, I thought why not combine it with the insert statement, there by the intermediate result from selection need not be stored, which would avoid 2 disk I/O of entire table for each intermediate table (which would be huge!).

Other minor optimizations:

- In single pass algorithms like Projection, Deletion, I used all the blocks in memory except one to fetch the blocks from disk, and the processed result is then stored in the temporary block till it's full then I do in-place replacement of block in memory, so that I flush the maximal number of blocks at once.
- Wherever multiple relations' read/write to disk were involved, I made sure each relation gets equal amount of memory for maximal performance.

Source code:

- I am including the source code with entire eclipse workspace, you may import the workspace or the project into your eclipse workspace to view the source.