CSCE 608 - 601 Database Systems

Fall 2017, Course Project #2

Due Date: 12/05/2017

Team:

Geethik Narayana Kamineni – 825008990

Email ID: gn1581@tamu.edu

Prafulla Chandra Munugoti – 825005844

Email ID: prafulmunugoti@tamu.edu

Language used: C++

Specification:

The aim of this course project #2 is to design and implement a simple SQL (called Tiny-SQL) interpreter. This SQL Interpreter should accept SQL queries that are valid in terms of Tiny-SQL grammar, execute queries, and output results.

SQL 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. This should optimize of the selection operations in the tree.

• A physical query plan generator, which converts optimized logical query plans into executable physical query plans. This phase also includes any possible physical query plan optimization. This should at least optimize the join operations.

• 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 Storage Manager, which simulates computer disks and memory.

Procedure:

To meet the above requirements, a typical SQL Interpreter contains functionality to prepare a collection of efficient algorithms and operations in relational algebra.

In doing so, the SQL Interpreter must parse the input program using a Parser or similar application. Once keywords and Tokens are correctly identified and stored. a preprocessor can then validate the program using a combination of view processing and semantic checking techniques based on the expected language criteria.

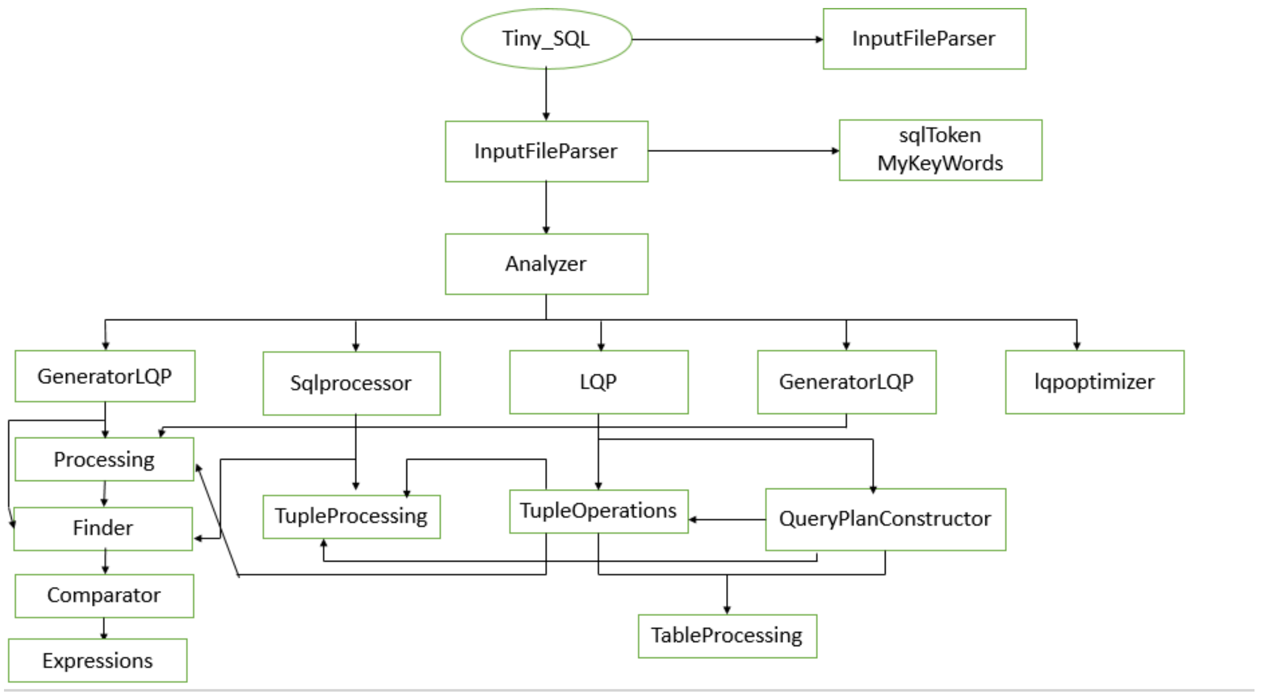
We will be constructing the parse tree based on the SQL Query that is given as input. Once a parse tree has been constructed, the SQL Interpreter can then formulate a logical query plan (LQP). This phase may include a series of optimizations based on relational algebra, logic laws, and also a reduction in size of any intermediate results.

After the LQP-phase is complete, based on the LQP, the SQL Interpreter will then generate a physical query plan (PQP). This component of the SQL Interpreter is responsible for handling optimizations at the physical storage components of the system, including main memory and disk.

To account for the disadvantages of both--memory is fast, but small and volatile, while disks are just the opposite--the LQP-to-PQP converter will make necessary decisions based on the layout of the available data storage hierarchy and enforce efficient storage and retrieval algorithms--e.g., based on cost estimations of 2 certain disk operations--and also utilize data structures and computational modes in order to maintain efficiency.

Architecture of the Software:

Flow Diagram:



Explanation:

Comparator.cpp: This program handles all the comparison operators like <, >, =.

Finder.cpp: This program handles all the conditional operators like AND, OR and NOT.

sqlprocessor: This program has the methods to deal with all the SQL operations like table creation, tuple insertion, selecting the table, dropping the table and delete the table.

sqlToken: This program declares variables to hold the tokens and their data types (INT, and STR)

Expressions: This program is used to evaluate the basic arithmetic operations like +, -, *, /.

InputFileParser: This program parses the input file line by line and returns the complete file to the calling function.

LexicalAnalyzer: This program analyses the input file by returned by the input file Parser to tokenize the input.

LQP: This program helps creating a logical Query Plan like, SelectQP, ProjQP, JoinQP and multiple combinations of them.

GeneratorLQP: This program helps in generating the query plan that is generated in the LQP program.

Lqpoptimizer: This program helps in optimizing the QueryPlan.

Tiny_SQL: This program handles all the input/output operations like inputting the SQL Query, conditions to check for it.

Analyzer: This program is to parse the statements from input file by subdividing each statement into tokens and return them to a variable that is passed further.

Processing: This program performs Processing Operations like Projection Operation, Join operation and Order Operation.

QueryPlanConstructor: This program is for constructing the QueryPlan.

TupleOperations: This program is to perform the operations on the tuples.

Data Structures used:

Vectors, Tree Structure after parsing the Input SQL Query.

Algorithms used:

One pass algorithms in Tuple Operations such as projection, selection, product, join, duplicate elimination, and sorting as well as the two-pass algorithms for join, duplicate elimination, and sorting.

Experiments and Results:

a.) 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.

Note: Every table column1 – # of Tuples, column2 – Disk IO's Colm3 – Exec Time(ms)

Input file: TinySQL-TextLINUX.txt

SELECT * FROM course		
No of Tuples	***Disk IO's***	***Execution Time(ms)***
2	2	149.26
6	6	447.78
50	60	4477.8
SELECT sid, grade FROM course		
50	60	4477.8

DELETE FROM course WHERE grade = "E"		
6	7	522.41
SELECT sid, course.grade FROM course		
50	60	4477.8
SELECT * FROM course WHERE exam > 70		
50	60	4477.8
SELECT DISTINCT course.grade, course2.grade FROM course, course2 WHERE course.sid = course2.sid		
73	72	5256.43
SELECT * FROM r, s, t WHERE r.a=t.a AND r.b=s.b AND s.c=t.c		
122	1715	127778

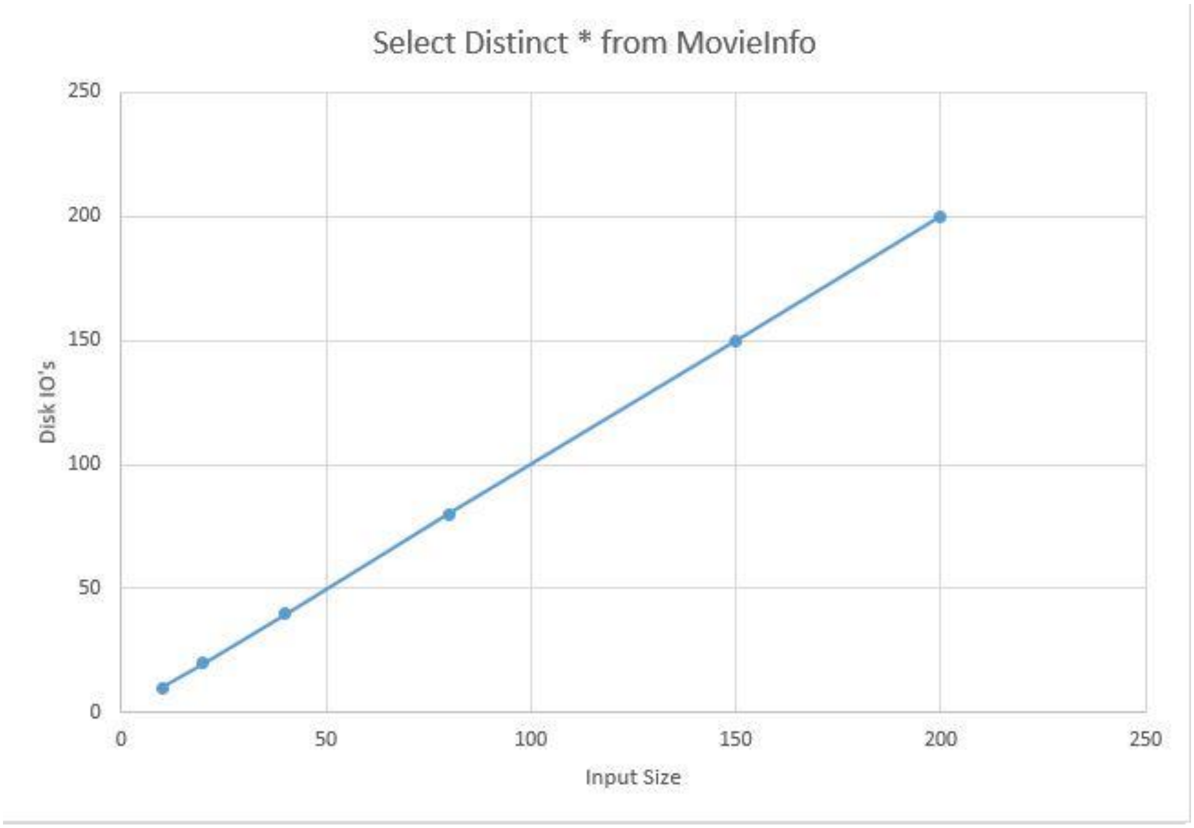
SELECT DISTINCT grade FROM course		
No of Tuples	***Disk IO's***	***Execution Time(ms)***
50	60	4477.8
SELECT * FROM course ORDER BY exam		
50	60	4477.8
SELECT * FROM course WHERE exam = 100		
50	60	4477.8
SELECT * FROM course WHERE exam = 100 AND project = 100		
50	60	4477.8
SELECT * FROM course WHERE NOT exam = 0		
50	60	4477.8
SELECT * FROM course, course2		
53	62	4616.43
73	72	5256.13
SELECT * FROM t1, t2, t3, t4, t5, t6		
18	896	66868.5

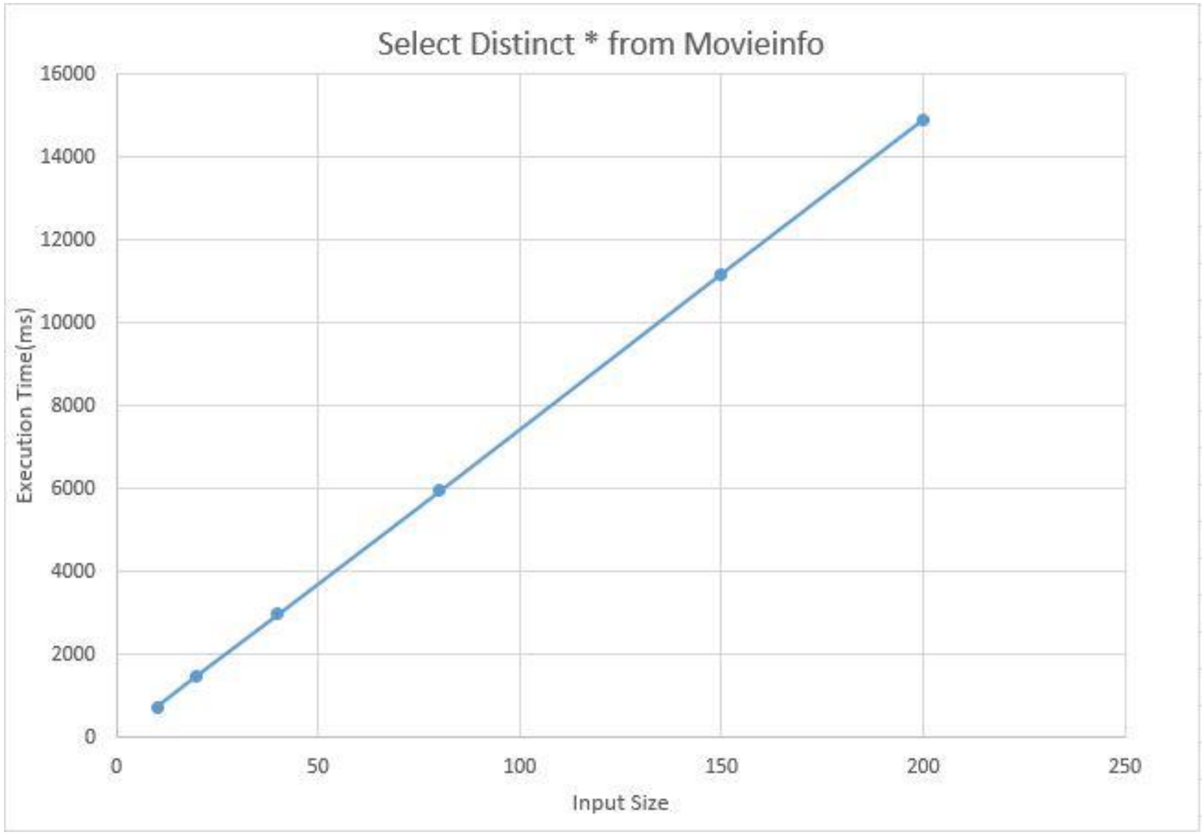
Total Time taken for all the queries in the file: 557.312 seconds b.) The running time and the number of disk I/O’s required to execute sample queries. Collect several measurements of running time versus data size, and disk I/O’s versus data size. Input file used is SongsInfo attached in the Zip Folder under Testing Folder.

SELECT DISTINCT * FROM movieinfo		
No of Tuples	***Disk Io's***	***Execution Time(ms)***
10	10	746.3
20	20	1492.6
40	40	2985.2
80	80	5970.4
150	150	11194.5
200	200	14926
SELECT * FROM movieinfo WHERE movid = 200		
220	220	16418.6
DELETE FROM movieinfo WHERE year = 2000		
220	415	30971.5

SELECT * FROM movieinfo WHERE movid > 5		
No of Tuples	***Disk Io's***	***Execution Time(ms)***
10	10	746.3
20	20	1492.6
40	40	2985.2
80	80	5970.4
150	150	11194.5
200	200	14926

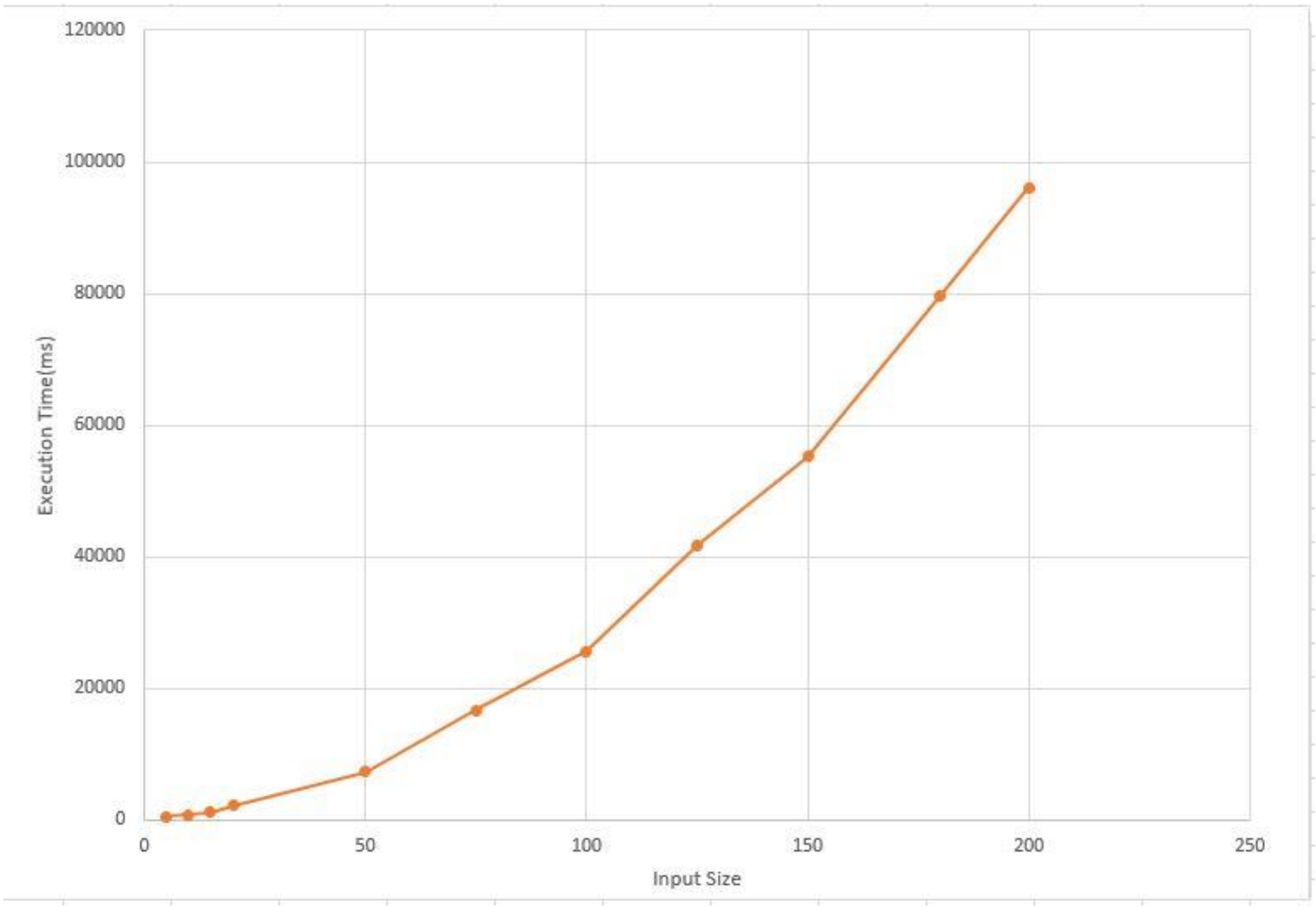
Comparison Graphs: Graphs remain same for all the above as similar data is present.

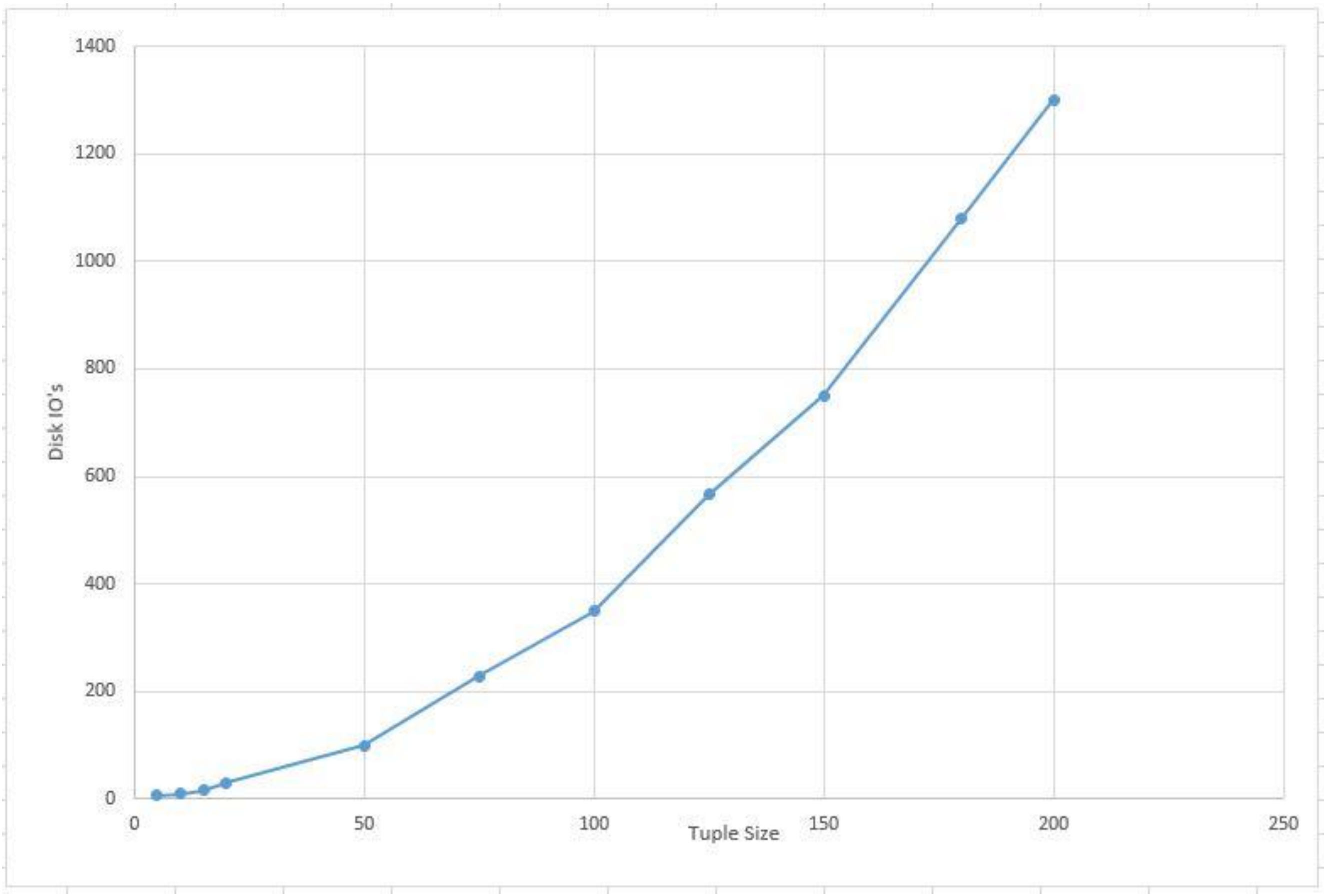




Input file Name: Address

SELECT * FROM customer, address		
Input Size	***Disk I/O's***	***Execution Time (ms)***
5	6	426.52
10	10	703.78
15	16	1119.67
20	30	2153.86
50	100	7229.14
75	228	16664.8
100	350	25652.8
125	567	41719.9
150	750	55270.9
180	1080	79750
200	1300	96083.6





Changes to Library:

1. The following changes were made to the provided StorageManager library:
C++11 requires constexpr and hence 3 lines Disk.h in Storage Manger library have been changed from :

//static const double avg_seek_time=6.46;
//static const double avg_rotation_latency=4.17;
//static const double avg_transfer_time_per_block=0.20 * 320;
 to

#define avg_seek_time 6.46
#define avg_rotation_latency 4.17
#define avg_transfer_time_per_block 64

2. #define NUM_OF_BLOCKS_IN_MEMORY 300 is used to measure algorithm performance on 200 tuples or more, without this facing an Segmentation Fault (core dumped issue) Works fine for the normal cases.