Satya Aditya Praneeth Emani - 524002563
Cameron Smith - 520002417
Dr. Jianer Chen
CSCE-608: Database Systems
06 December 2016

Report - Project #2

Introduction

Database management system

A database management system (DBMS) is a software application that translates user database queries into executable machine programs. In order to accomplish this, the DBMS structure necessarily requires a set of specified functionality and behaviors. This includes the ability to efficiently handle, as well as manipulate data stored in hierarchical memory structures, the capability to translate input database programs (e.g., queries) into internal representations and data structures, support to enforce data consistency, and also a means to ensure reliability.

Components of a DBMS

In order to satisfy these requirements, a typical DBMS contains functionality to prepare a collection of efficient algorithms and operations in relational algebra. In doing so, the DBMS must first parse the input program using a language compiler or similar application. Once keywords and tokens have been correctly identified, a preprocessor can then validate the program using a combination of view processing and semantic checking techniques based on the expected language criteria.

Once a parse tree has been constructed, the DBMS 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 DBMS will then generate a physical query plan (PQP). This component of the DBMS 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

certain disk operations--and also utilize data structures and computational modes in order to maintain efficiency.

The Project

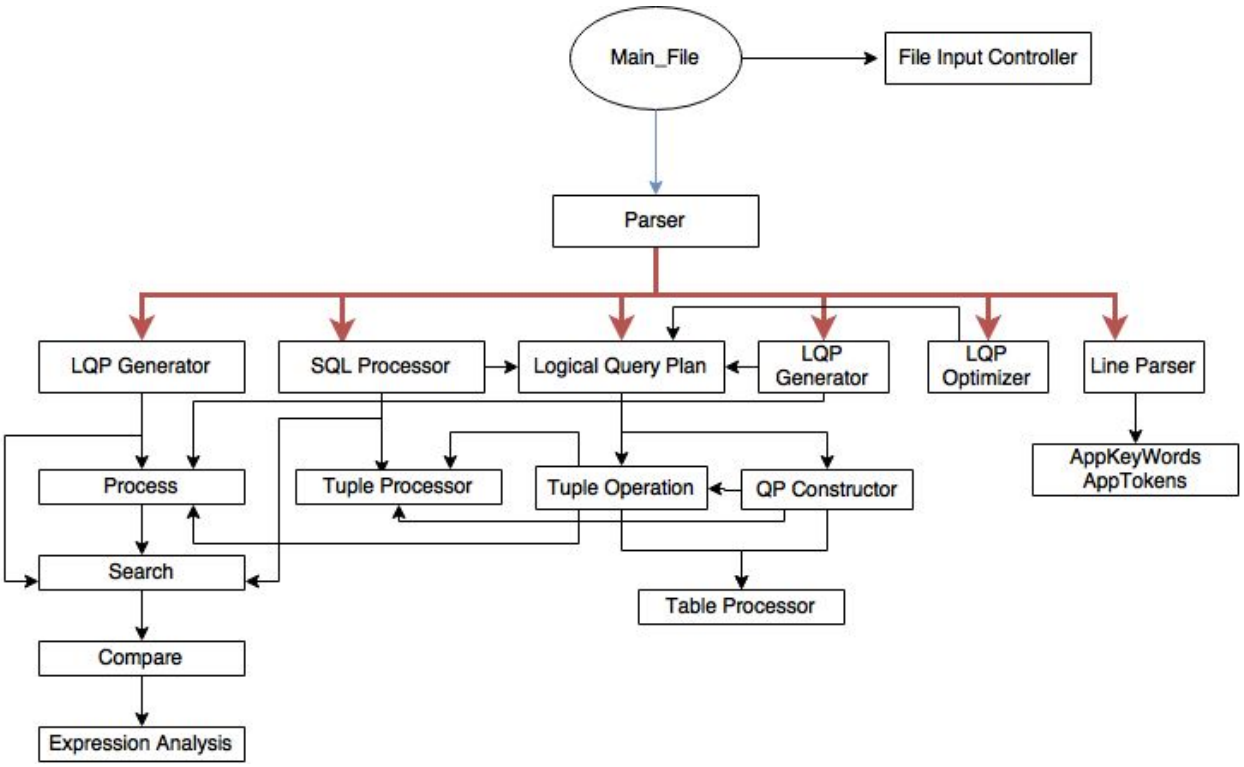
Description

The specification for this project was to design and implement a query interpreter based on the grammar of Tiny-SQL, a simplified version of the Structured Query Language (SQL), which is a widely-used language for relational database management and data manipulation.

Components

- AppTokens - This class declares variables to hold the token and its data type
- Compare - Handles all scenarios involving comparison operators
- ExpressionAnalysis - This class evaluates expressions such as (+.-,/,*)
- FileInputController - This class reads the input file line by line and returns the complete file to the calling function.
- LineParser - Analyzes the input line and tokenizes the input
- LogicalQueryPlan - Creates a simple LQP
- LQPGenerator - This class generates the Optimized Query Plan
- LQPOptimizer - This class optimizes the LQP
- Main_File - This class handles all I/O Operations
- Parser - Used to parse the statements from input file by subdividing each statement into tokens.
- Process - This class performs Processing Operations
- QPConstructor - This class constructs the query plan
- Search - This class performs logical operations such as AND, OR and so on.
- SQLProcessor - All the SQL operations are performed by the methods of this class
- TupleOperations - This class performs operations on Tuples
- TupleProcessing - This class performs Tuple Processing.

Architecture of Software



Experimental results

Both team members were able to successfully transform the larger portions of their datasets used in the first project into a series of test cases for Project #2. In these test cases, we observed that as the number of tuples per table expanded (this was done using a series of 'INSERT INTO …' statements), as expected the time out system took to successfully produce output for subsequent 'SELECT …' queries grew rapidly as a result. As a specific example, the test scripts we created each included the population of a table of up to 200 tuples.

Below is a table showing the number of disk I/Os as well as the time it took to execute certain commands.

Command	**Disk I/O Operations**	**Time (ms)**
CREATE TABLE	0	0
INSERT	1	74.63
SELECT (1 TUPLE)	1	74.63
SELECT (2 TUPLES)	2	149.26
DROP TABLE	0	0
SELECT * FROM course	60	4477.8
SELECT sid, grade FROM course	60	4477.8
SELECT DISTINCT grade FROM course	300	21241
SELECT * FROM course ORDER BY exam	300	21241
SELECT * FROM course WHERE exam = 100	60	4477.8
SELECT * FROM course WHERE exam = 100 OR homework = 100 AND project = 100	60	4477.8

Total Time Taken in seconds: **497.56**

Reflections

Challenges Cameron faced

Early on, I struggled with the challenge of parsing the input programs (queries). I would have liked to have incorporated the use of language parsers such as Lex and YACC. Having initially attempting to write this portion of the project by hand, I have come to see the value of such language parsing tools. However, I found that the complexity of learning how to use these tools in the given amount of time for this project outweighed the time it would have taken me to code custom parsing subroutines.

Challenges Satya faced

I took considerate amount of time in the development of Logical Query Plan, Tuple Operations and Parse tree structure. My major challenge was in integrating the whole application as one.