# CS39202 - DBMS Laboratory

## Term Project Report

***Group Name: Outer Rim***
**1) Ashwani Kumar Kamal - 20CS10011**
**2) Archit Mangrulkar - 20CS10086**
**3) Hardik Pravin Soni - 20CS30023**
**4) Shiladitya De - 20CS30061**
**5) Sourabh Soumyakanta Das - 20CS30051**

## §1 Project Description

**Name:** Rule based query rewriting
**Description:** Specify some fixed rules for query optimization. Write a query rewriter for simple queries which takes as input a relational algebra query and returns its optimal version.

## §2 Introduction

Query optimization is a critical step in relational database management systems (RDBMS). The process involves identifying the most efficient way to execute a query and optimizing it to minimize execution time and resource usage. In this proposal, we outline a project to specify fixed rules for query optimization and develop a query rewriter for simple queries. The project's objective is to provide a tool that takes as input a relational algebra query and returns its optimal version by applying the specified query optimization rules.

## §3 Project Scope and Objectives

The project's scope is to specify fixed rules for query optimization and develop a query rewriter for simple queries. The project will involve the following objectives:

- Specification of fixed rules for query optimization.
- Design and implementation of a query rewriter tool.
- Testing the query rewriter on a suitable testset.
- Documentation of the design, implementation, and representing the results in a comprehensive report.

## §4 Project Deliverables

The project contains the following:

- A report containing the detail of execution of the work.

- A presentation of the same.
- A website which presents the query rewriter tool.
- Link of the codebase (Github)

## §5 Methodology

In this section, we are describing of methodology approaching and executing a solution to the problem. This is as follows:

### §5.1 Specifying syntax for giving relational algebra query

Firstly, we defined a proper syntax to give relational algebra query input by the user. The syntax finalized by us is as follows:

- $\sigma$ : `SELECT`
- $\prod$ : `PROJECT`
- $\bowtie$ : `JOIN`
- $\times$ : `PROD`
- $\bigcup$ : `UNION`
- $-$ : `DIFF`
- $\bigcap$ : `INTERSECT`
- $\wedge$ : `AND`
- $\vee$ : `OR`

**Example of full fledged queries :**
Actual Query: $\sigma_{(a \wedge b)}(T)$
Query according to the rules given above: `SELECT [a AND b] (T)`

Actual Query: $\prod_{(a)}(\prod_{(b)}(\sigma_{(c \wedge d)}(T))$
Query according to the rules given above: `PROJECT [ a ] (PROJECT [ b ] (SELECT [ c AND d ] (T)))` etc.

### §5.2 Rules to be used for query optimization:

Only those rules are considered which are heuristic based and not based on the actual table statistics (cost based ones). These rules are as follows:

- $\sigma_{(a \wedge b)}(T) = \sigma_{(a)}(\sigma_{(b)}(T))$
- $\prod_{(a_1)}(\prod_{(a_2)}(...(\prod_{(a_n)}(T)...)) = \prod_{(a_1)}(T)$
- $\sigma_{\theta}(T_1 \times T_2) = T_1 \bowtie_{\theta} T_2$
- $\sigma_{\theta_1}(T_1 \bowtie_{\theta_2} T_2) = T_1 \bowtie_{(\theta_1 \wedge \theta_2)} T_2$
- $\sigma_{\theta}(T_1 - T_2) = \sigma_{\theta}(T_1) - \sigma_{\theta}(T_2)$
- $\prod_{\theta}(T_1 \cup T_2) = \prod_{\theta}(T_1) \cup \prod_{\theta}(T_2)$

## §5.3 Code

We have used a compiler based approach to solve this problem. Since each of the optimizations are based on some grammar or a specified structure so using flex - bison to parse the queries then using the tree generated to optimize the queries based on the rules described above. The code for the query optimizer is divided into following three parts which are:

### §5.3.1 Lexer

The flex tool is used to get the tokens of input query like `SELECT`, `PROJECT`, conditions etc. based on regex expressions defining them and sending them to bison which then parses it.

### §5.3.2 Bison

Bison is an effective tool to generate a parse tree based on a grammar provided to it. It is a LR parser which does a bottom up parsing and sends the structures created to the functions of Query Optimizer which were called from it.
The grammar used is as follows:

```
%union
{
    char* str;
    Node* node;
}

%token AND OR SELECT PROJECT JOIN UNION INTERSECT DIFF PROD LSQ RSQ LPAR RPAR
%token <str> WORD

%start result

%type<node>
    expression
    table
    term

%%

result :
        | result table
        {
            // printTable($2);
            revert($2);
        }
;

table : term
        { $$ = makeTree($1); }
        | LPAR table RPAR
        { $$ = $2; }
        | SELECT LSQ expression RSQ LPAR table PROD table RPAR
        { $$ = joinTree($3, $6, $8); }
        | SELECT LSQ expression RSQ LPAR table JOIN LSQ expression RSQ table RPAR
        { $$ = joinTree(appendAndNode($3, $9), $6, $11); }
        | SELECT LSQ expression RSQ LPAR table DIFF table RPAR
        { $$ = diffTree($3, $6, $8); }
        | SELECT LSQ expression RSQ LPAR table RPAR
        { $$ = makeSelectTree($3, $6); }
        | PROJECT LSQ expression RSQ LPAR table RPAR
        { $$ = makeProjectTree($3, $6); }
        | PROJECT LSQ expression RSQ LPAR table UNION table RPAR
```

```
        { $$ = unionTree($3, $6, $8); }
;

expression: term
            { $$ = $1; }
            | expression AND term
            { $$ = makeAndNode($1, $3); }
;

term:   WORD
        { $$ = makeTermNode($1); }
        | term OR WORD
        {
            string s = ($1)->content + " OR " + $3;
            $$ = makeTermNode(s);
        }
;
```

### §5.3.3 Query Optimizer

Various functions that are implemented are :

`void revert(Node* node):` This function recursively prints the result by traversing the parse tree from top to bottom of the tree.

`void printTable(Node* node):` This function recursively prints the content of the tree nodes thereby depicting a parse tree.

`Node *makeTermNode(string s):` This function generates a node which is of the type WORD_ and is a leaf node.

`Node *makeAndNode(Node* left, Node *right):` This function generates a node of the type AND_ and joins a list of AND attributes with another attribute using AND.

`Node *parse(Node* node):` This function returns the node of the new tree which is generated after parsing the AND attributes using get_and function.

`Node *get_and(Node* node, Node* res):` This function recursively generates a new parse tree after processing the AND attributes.

`Node *appendAndNode(Node* left, Node *right):` This function merges two AND_ trees and generates a single AND_ tree using appendNode function.

`Node *appendNode(Node* left, Node *right):` This function recursively goes to the far left of the second tree where it appends the first tree after performing suitable modifications.

`void get_str(string s, Node* node):` This function is used to generate a string containing all the AND attributes in properly formatted manner.

`Node *makeTree(Node* left):` This function generates a TABLE_ node which is used to append a table to the parse tree structure.

`Node *makeSelectTree(Node* left, Node *right):` This function generates a node of type STABLE_ which denotes the node referring to a SELECT query.

`Node *makeProjectTree(Node* left, Node* right):` This function generates a node of type PTABLE_ which denotes the node referring to a PROJECT query.

`Node *get_table(Node *tree):` This function recursively searches for a node which is not of type PTABLE_.

`Node *gen_copy(Node *node):` This function recursively generates a copy of the tree whose root node is provided.

`Node *joinTree(Node* attr, Node* table1, Node* table2):` This function generates a node of type JTABLE_ which denotes the node referring to a JOIN operation.

`Node *diffTree(Node* attr, Node* table1, Node* table2):` This function generates a node of type DTABLE_ which denotes the node referring to a DIFFERENCE operation.

`Node *unionTree(Node* attr, Node* table1, Node* table2):` This function generates a node of type UTABLE_ which denotes the node referring to a UNION operation.

`Node *intersectTree(Node* table1, Node* table2):` This function generates a node of type ITABLE_ which denotes the node referring to an INTERSECT operation.

# §6 Results

## §6.1 Test Queries and their Results

**Query:** `SELECT [ A1 AND B5 AND C5 ] ((T))`
**Result:** `SELECT [ A1 ] (SELECT [ B5 ] (SELECT [ C5 ] (T)))`

**Query:** `SELECT [ A2 ] (SELECT [ B1 AND C7] (T))`
**Result:** `SELECT [ A2 ] (SELECT [ B1 ] (SELECT [ C7 ] (T)))`

**Query:** `PROJECT [ B1 ] (PROJECT [ A2 ] (T))`
**Result:** `PROJECT [ B1 ] (T)`

**Query:** `SELECT [ A1 AND B5 ] (PROJECT [ B1 ] (PROJECT [A2] (T)))`
**Result:** `SELECT [ A1 ] (SELECT [ B5 ] (PROJECT [ B1 ] (T)))`

**Query:** `PROJECT [ B1 ] (PROJECT [ A2 ] (SELECT [ A1 AND B5 ] (T)))`
**Result:** `PROJECT [ B1 ] (SELECT [ A1 ] (SELECT [ B5 ] (T)))`

**Query:** `SELECT [ A3 ] (T1 DIFF T2)`
**Result:** `SELECT [ A3 ] (T1) DIFF SELECT [ A3 ] (T2)`

**Query:** `PROJECT [ A5 ] (T1 UNION T2)`
**Result:** `PROJECT [ A5 ] (T1) UNION PROJECT [ A5 ] (T2)`

**Query:** `SELECT [ A3 ] (PROJECT [ A5 ] (T1 UNION T2) DIFF T2)`
**Result:** `SELECT [ A3 ] (PROJECT [ A5 ] (T1) UNION PROJECT [ A5 ] (T2)) DIFF SELECT [ A3 ] (T2)`

**Query:** `SELECT [ A4 ] (T1 PROD T2)`
**Result:** `T1 JOIN [ A4 ] T2`

**Query:** `SELECT [ A7 ] (T1 JOIN [ B3 ] T2)`
**Result:** `T1 JOIN [ A7 AND B3 ] T2`

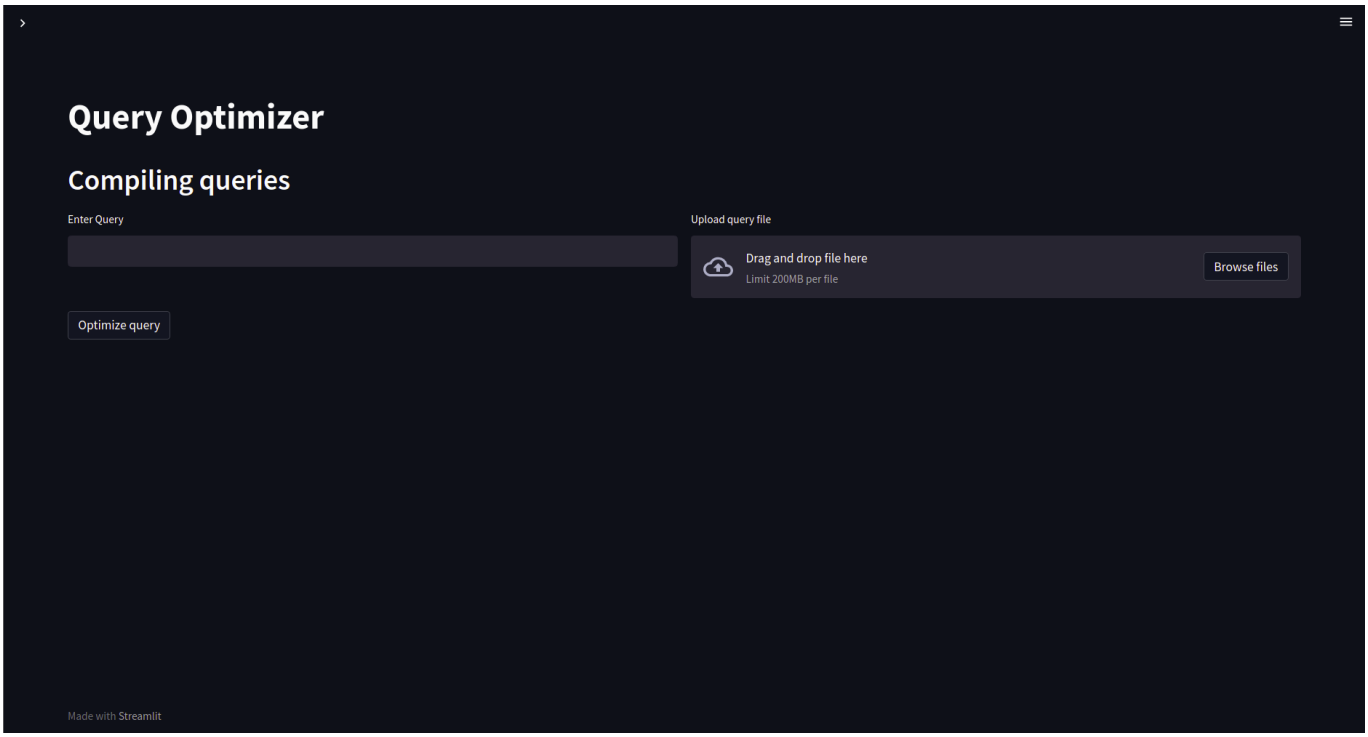
**Query:** `PROJECT [ A5 ] (SELECT [ A3 ] (T1 DIFF T2) UNION T2)`
**Result:** `PROJECT [ A5 ] (SELECT [ A3 ] (T1) DIFF SELECT [ A3 ] (T2)) UNION PROJECT [ A5 ] (T2)`

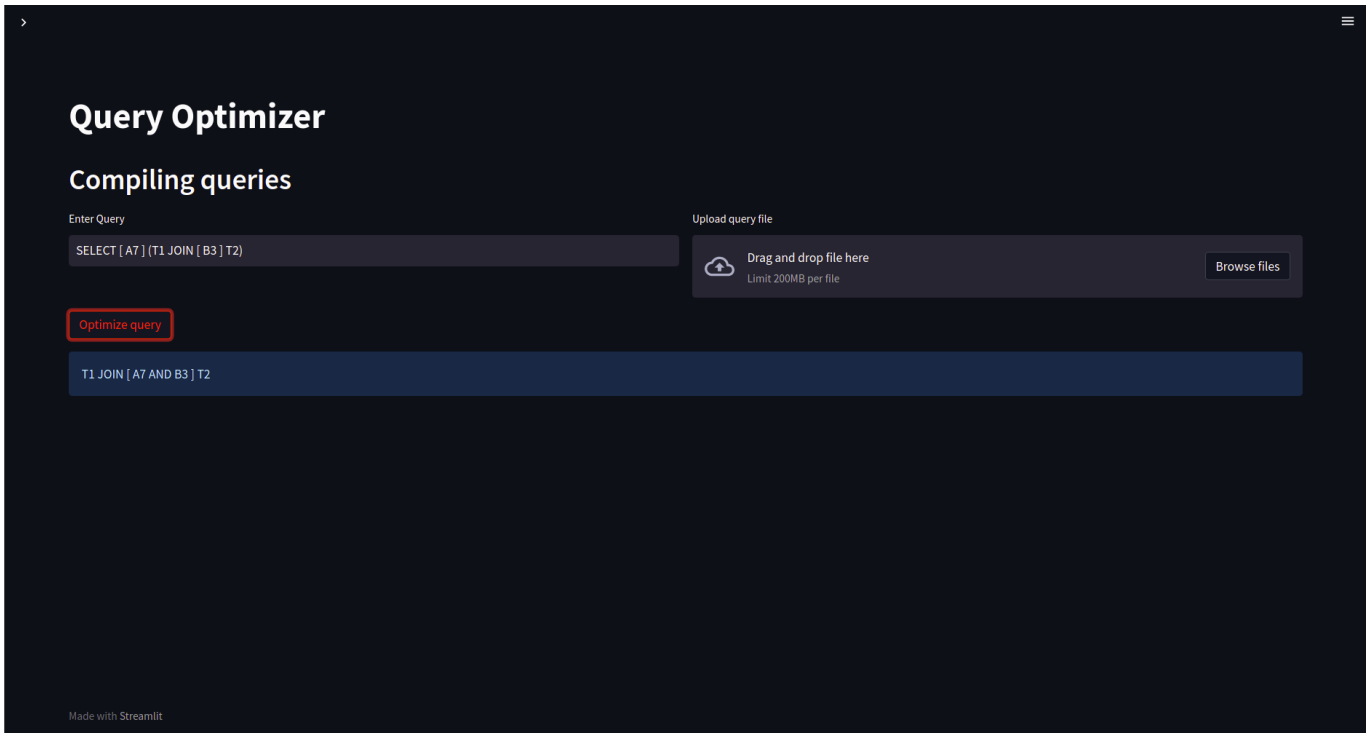
**Query:** `SELECT [ A3 ] (SELECT [ A1 AND B5 ] (T) DIFF T2)`
**Result:** `SELECT [ A3 ] (SELECT [ A1 ] (SELECT [ B5 ] (T))) DIFF SELECT [ A3 ] (T2)`

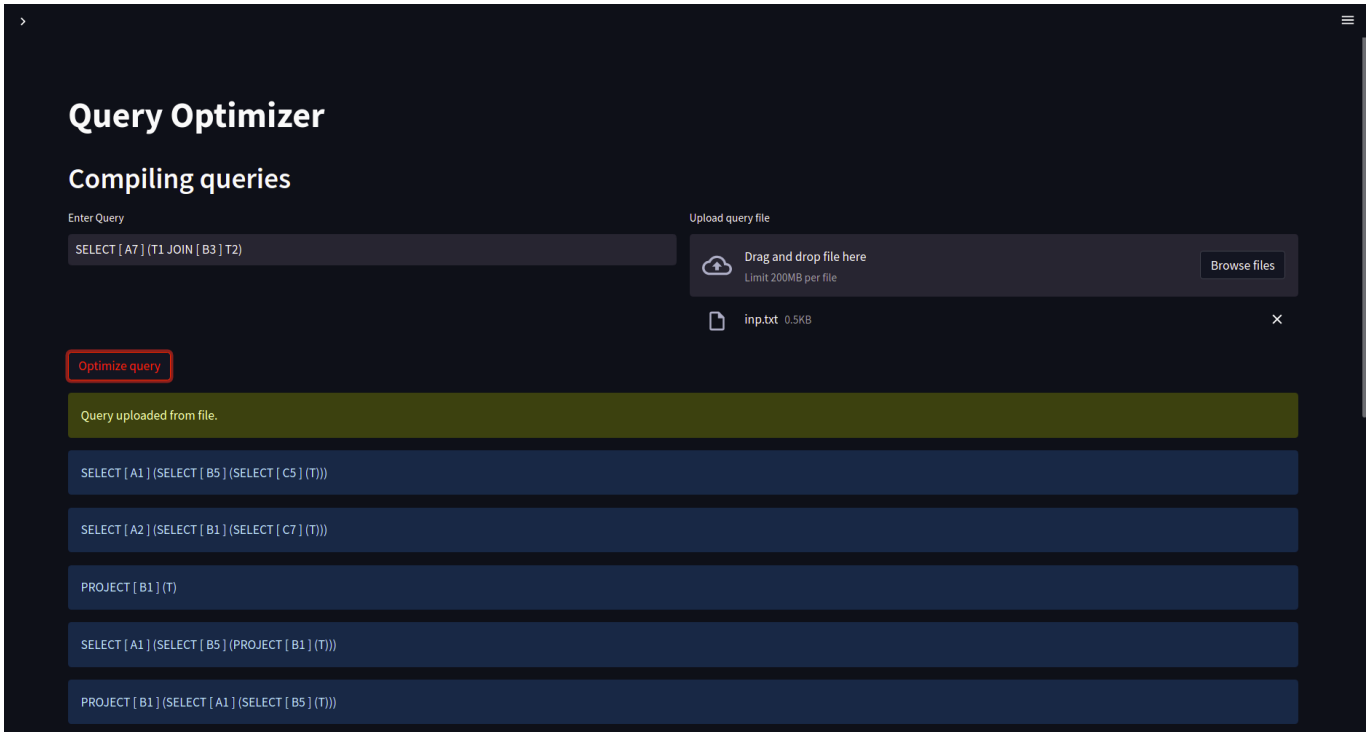
## §6.2 Frontend Application

A frontend interface has been created which has an input text box for testing queries on the go and option to upload a file with queries separated by newlines.

Some screenshots of the same are attached below






## §7 References

1. Chaudhuri, S. (1998). An overview of query optimization in relational systems. ACM SIGMOD Record, 27(2), 34-49.

2. Graefe, G. (1995). Volcano-an extensible and parallel query evaluation system. IEEE Transactions on Knowledge and Data Engineering, 7(4), 563-575.

3. Selinger, P. G., Astrahan, M. M., Chamberlin, D. D., Lorie, R. A., & Price, T. G. (1979). Access path selection in a relational database management system. In Proceedings

## §8 Appendix

**Link to codebase (github):** https://github.com/outer-rim/Query-Optimiser

**Link to the website:** http://20.244.34.245:8501