

# A2 - Report

## Approach :

My application takes the given relational schema and relational algebra query and gives back an optimized query. A visual representation of the original and optimized query tree is also outputted, along with the associated cost of each query tree. This process can be broken down into the following steps:

### 1. Parse & store the incoming relational schema and relation algebra query

In order to parse the input file, I simply read in the file line by line using the getline function. Depending on the type of statement ("TABLE", "OP", "Cardinality", "SIZE" etc.), the program will obviously parse differently. Please use the formatting mentioned in the formatting section later on as this allows the program to properly parse the content of the input file. I have tried to make the parsing as robust **as possible**, however since this is not the main focus of this assignment, please use the suggested formatting to ensure you are using my program properly. While parsing, the program also stores the information (tables, operations, index statistics) into objects for later use.

### 2. Calculate cost of original query

Once we have all the necessary information stored, we compute the cost of the query. Depending on the type of operation, we calculate cost differently:

- **SELECTION, PROJECTION**

For selection and projection, if the input to the operation is a base table, we need to read the entire relation and thus we must read all the files (npages) of the base table (plus the cost of writing to a temp). To calculate the number of tuples in the result, we can use the provided RFs. If the selection uses '>', then my application calculates the RF as shown in lectures:

- Term  $col > value$  has RF
  - $(High(I)-value)/(High(I)-Low(I))$

If the input to the operation is NOT a base table, we can assume that the unary operation is performed on-the-fly (as shown in Module 4).

- **JOIN**

For join, if there is not an index on the inner relation my application uses  $NPages(Outer\ Relation) + NTuples(Outer\ Relation) * NPages(Inner\ Relation)$  to get cost. If there is an index on the inner relation of the join, my application uses  $NPages(Outer\ Relation) + NTuples(Outer\ Relation) * 1.2$ . This is under the assumption that on average it will take 1.2 I/Os to find a matching tuple, as per Module 4.

The total cost of the query is simply calculated as the sum of all the costs for each operation.

### **3. Build query tree for original query**

For each operation and base table, a node is created. Then I simply link the nodes together based on the operations identified in the input file. Join operations are linked to two child nodes, and selection/projection operations are linked to one child node.

### **4. Build optimized query tree**

As mentioned in the assignment details, we can assume that a pipelined approach is always preferred. Hence this process of my application converts the original query tree into a query tree which takes advantage of pipelining. This is done by first pushing selections and projections up, to help us later ensure that the property of a left-deep tree can be obtained. Once they have been pushed up, the application looks at each join operation in the tree and makes sure that the right child of the join is a base table. Because in the earlier step we pushed up selections/projections, the worst case we can have is that the right child of a join operation, is another join operation. To convert this into a left-deep tree, the application first checks the columns on which we are joining. It then pushes the child join above the parent join. The table which was required in the original join (based on the join condition), becomes the new right child of the original join. The top join's left child is the original join, and the right child is the base table which

was not required for the original join. Take a look at the sample cases provided to get a visual understanding of what happens.

### **5. Calculate cost of optimized query**

Since we have created a left-deep tree, we can evaluate joins using pipelining. The selections and projections will simply be performed on-the-fly. The cost of join is calculated using the same method mentioned in step 2. Please refer to that step to see the details. The overall cost is significantly reduced as we do not have to materialize the left child (outer table). To get the total cost, we simply sum the cost of the joins.

### **6. Produce output**

The application prints the original tree and its associated cost. It then prints the optimized tree and its associated cost.


Note: Since we are assuming that we always prefer a pipelined approach, my application is built to provide such an optimized version. Hence there may be cases in which the I/O cost is more for the pipelined approach, but the optimization is that because it is optimized we don't have to wait for materialization of temporary operations.

## How to use the application :


Please run this application on the student environment.

After downloading **A2.cpp** from the learn drop-box:

1. Run the following command to get the **A2** executable

```
 g++ -g -std=c++14 -Wall A2.cpp -o A2
```

2. Run the application by providing it a valid input file

```
 ./A2 sampleTest.txt
```

Please let me know if you are facing any issues getting it to run.

## Input File Requirements :

In order to properly use this application, please make sure the input is formatted properly. Specifically, ensure the following:

- The spacing in the input file matches the sample statements given below:

Table Creation:

```
TABLE DEPARTMENT(Dname,Dnumber,Mgr_ssn,Mgr_start_date,PRIMARY KEY(Dnumber));
```

Foreign Key Creation:

```
FOREIGN KEY (EMPLOYEE(Dno) REFERENCES DEPARTMENT(Dnumber));
```

### Selection Operation:

```
OP1 = EMPLOYEE SELECTION Dno=5
```

### Projection Operation:

```
OP2 = OP1 PROJECTION Bdate,Dno
```

### Join Operation:

```
OP3 = OP2 JOIN DEPARTMENT ON Dno=Dnumber
```

### Cardinality (Table):

```
Cardinality(EMPLOYEE) = 10000
```

### Size (Table):

```
SIZE(DEPARTMENT) = 1
```

### Cardinality (Index):

```
Cardinality(Dnumber in DEPARTMENT) = 10
```

### Size (Index):

```
SIZE(Ssn in EMPLOYEE) = 100
```

### RF:

```
RF(Fname in EMPLOYEE) = 0.1
```

### Range:

```
Range(Dno in EMPLOYEE) = 1,10
```

- Please ensure that all columns, indexes have their necessary statistics represented in the input file as well, since this directly impacts cost

## Examples :

**\*\* The input files are too long to show in the report, hence I will only show the output here. Please refer to the .txt files attached on the learn submission to see the input file used for each example \*\***

Example 1 Output (subTest1.txt) :

```
-----  
| Query Tree |  
-----  
  
RESULT  
├─ OP1  
│   ├── DEPARTMENT  
│   └── EMPLOYEE  
└─ OP2  
    ├── DEPARTMENT  
    └── DEPT_LOCATIONS  
  
Cost: 13242 I/Os  
  
-----  
| Optimized Query Tree |  
-----  
  
OP1  
├─ DEPARTMENT  
└─ RESULT  
    ├── EMPLOYEE  
    └─ OP2  
        ├── DEPARTMENT  
        └── DEPT_LOCATIONS  
  
Cost: 1122 I/Os
```

## Example 2 Output (subTest2.txt)

-----  
| Query Tree |  
-----

```
RESULT
├── OP3
│   └── OP2
│       ├── OP1
│       │   └── DEPT_LOCATIONS
│       └── DEPARTMENT
└── EMPLOYEE
```

Cost: 21003 I/Os

-----  
| Optimized Query Tree |  
-----

```
OP3
└── OP1
    └── OP2
        ├── DEPT_LOCATIONS
        └── RESULT
            ├── DEPARTMENT
            └── EMPLOYEE
```

Cost: 11001 I/Os

Example 3 Output (subTest3.txt):

```
-----  
| Query Tree |  
-----  
  
RESULT  
└─ OP3  
    └─ DEPARTMENT  
        └─ OP2  
            └─ OP1  
                └─ EMPLOYEE  
  
Cost: 1300 I/Os  
  
-----  
| Optimized Query Tree |  
-----  
  
RESULT  
└─ OP2  
    └─ OP3  
        └─ DEPARTMENT  
            └─ OP1  
                └─ EMPLOYEE  
  
Cost: 1001 I/Os
```

## Libraries required to use program:

There are **NO** external libraries required for this application.

## References:

I used C++ documentation to understand how to use the transform function from the C++ algorithm library.

<https://en.cppreference.com/w/cpp/algorithm/transform>