# Assignment 4.2: Query Compilation and Optimization

**Due Finals Week**

The goal of this assignment is to compile and optimize an input SQL statement, and then print the resulting, optimized query plan to the screen. I have actually implemented a parser and lexer for the subset of SQL that you need to consider, which is available for download. The first thing that you need to do is to download and look over the parser and lexer. If you are confused about lex and yacc, you can look over Dr. Dobra's slides at http://www.cise.ufl.edu/~adobra/lectures/lex_yacc.pdf. Or a simple Google search will prove very informative as well.

## The Parser

The parser that I have implemented takes a simple SQL statement, processes it, and puts the result into a set of data structures (several linked lists as well as two integer yes/no flags). The linked lists and flags that are constructed by the parser are declared in the file parser.y:

```
// these data structures hold the result of the parsing
struct FuncOperator *finalFunction;
struct TableList *tables;
struct AndList *boolean;
struct NameList *groupingAtts;
struct NameList *attsToSelect;
int distinctAtts;
int distinctFunc;
```

In order to access these data structures externally from the parser, simply use an `extern` declaration in the file that you want to access them from.

Basically, what you'll do in your main program is to parse the input stream via a call to `yyparse()`, and then when you want to do your compilation and optimization, you will access these data structures as needed. Look in parser.y to see exactly what each of the structures mean and the sort of data that they hold.

You should also look over the parser to see exactly what our version of SQL will look like. To try out a simple SQL statement, just compile the parser that I've supplied, and then run the resulting executable. Type in:

```
SELECT SUM DISTINCT (a.b + b), d.g
FROM a AS b
WHERE ('foo' > this.that OR 2 = 3) AND (12 > 5)
GROUP BY a.f, c.d, g.f
<control-D>
```

And the program will parse this statement and fill up the data structures!

## What You Need To Do

Since the parser is already written, you probably won't have to do much in terms of messing with parser.y (though undoubtedly there are things that I've done that you won't like, and will want to change). But the core of this assignment is to implement the recursive routine for enumerating all possible query plans, and selecting the best one in terms of the plan that creates the smallest number of intermediate tuples. You will use your stats code from the first part of A4 to guess the size of the intermediate relations that your query plan will produce. For simplicity, just go ahead and initialize your statistics object with all of the statistics that you were provided in the first part of A4 (you might just want to put these into a text file in the format used by your A4.1 statistics object, and then when your main program fires up, you read them in in order to perform the optimization).

In the end, what your recursive optimization routine will produce is a query plan, which should be represented/implemented as a tree-based, C++ data structure. There should be seven different node types that can appear in the tree, with one node type being associated with each relational operation from A3. Each node should contain the output schema for records that are being piped out of the operation, as well as any other operation-specific information that will be needed to actually invoke or call the relational operation. For example, in the case of a relational projection node, you will need to store the last three parameters to the `Project` operation in the node. In the case of a relational join node, you will need to store the last two parameters to the `Join` operation in the node. You do not actually need to store any of the pipes that will be used in the nodes, since these are implied by the structure of the tree. That is, if a join node is the child of a project node, then this implies that there is a pipe from the corresponding `Join` operation into the corresponding `Project` operation. These pipes will actually be created when you execute the query plan, which is the goal of the next (last) assignment.

In order to facilitate grading and debussing of this assignment, once your optimization routine produces its final query plan, what your main program should do is to perform an in-order traversal of the plan to print it out to the screen. Each of the nodes should be printed to the screen in something resembling the following format:

```
********
Join Operation
Input pipe ID 2
Input pipe ID 5
Output pipe ID 6
Output Schema:
     Att1: int
     Att2: int
     Att3: string
```

```
Join CNF:
     <output from CNF.Print ()>
********
```

Specifically, when a node is printed, its relevant pipes should be identified (associate a unique pipe ID with each pointer or link in the tree), the output schema should be printed, and the specific operation should be named (join, project, pipe select, etc.). Also, any important, operation-specific information needs to be printed to the screen. On an operation-by-operation basis, this is:

- In the case of a select on a pipe: the corresponding CNF
- In the case of a select on a file: the corresponding CNF
- In the case of a project: the attributes to keep
- In the case of a join: the corresponding CNF
- In the case of a sum: the corresponding function
- In the case of a group-by: the corresponding OrderMaker as well as the function

That’s it!