COMP 530 Assignment 5: SQL Semantic Checking

1. The Task

Your task is to take the SQL parser (written using `flex` and `bison`) that I have supplied you with, and then extend it so that it performs appropriate semantic checks on the input SQL. Specifically, you need to:

1. Make sure that there are no type mismatches in any expressions. For example, it is valid to compare integers and floating point numbers, but not integers and text strings. For another example, the only arithmetic operation that is valid on a text string is a "+" (which is a concatenation)... anything else should result in an error.
2. Make sure that all of the referenced tables exist in the database.
3. Make sure that all of the referenced attributes exist, and are correctly attached to the tables that are indicated in the query.
4. Make sure that in the case of an aggregation query, the only selected attributes (other than the aggregates) must be functions of the grouping attributes.

In the case that you find any errors, you should print out a descriptive and meaningful error message to the screen. You need only print one error per query. If there are multiple errors, there is no reason to print all of them.

2. Details, Details, Details

There I'm providing you with a lot of infrastructure that you can use to make this task easier. I describe some of this infrastructure now.

2.1 The Lexer and the Parser

The assignment ships with a lexer (developed using `flex`) and a parser (developed using `bison`). Note that since the parser uses `flex` and `bison`, you'll need to make sure that you have these tools installed on whatever platform you are doing your development.

You do not actually need to look inside of the lexer and the parser; it is possible to complete this assignment using the provided lexer and parser without understanding how they work and what they do (if you **do** want to understand how to use these tools, start by looking at `bison`; here is a nice slide presentation on the subject: http://www.eecg.toronto.edu/~jzhu/csc467/readings/csc467-bison-tut.pdf).

Regardless of whether you look carefully at `flex` and `bison`, you should look into the code for `main.cc`, which calls the parser. You will see that result of parsing the query is stored in the `SQLStatement *final`. Currently, all that happens is that the contents of this structure are printed out to the screen. To perform the required semantic checks, you'll need to process this structure, traversing it, and verifying that it has no semantic problems But you don't need to understand how the parsing works to do this; you just need to understand the `SQLStatement` class and the related classes. so that you can use it to perform semantic checking.

One thing to be aware of is that our parser will accept a relatively restricted version of SQL. Which is fine. For example, we require an alias for every table (that is, there must be as `AS` for every table in the `FROM` clause) and every reference to an attribute must use the dot notation. This simplifies things a bit. Further, we do not allow subqueries, we do not allow a "`SELECT *`", and our only aggregate functions are `SUM ()` and `AVG ()`.

2.2 Adding the Semantic Checks

You can go ahead and make changes to `main.cc`; in fact, you'll want to do this. However, don't write too much code there; a few lines added will do the trick. Most of your code will consist of changes (that is, additional methods) that you make to the various types in the file `ParserTypes.h` and `ExprTree.h`. `ParserTypes.h` contains the types that store the parsed query. `ExprTree.h` contains the types used to encode arithmetic and boolean expressions. You'll also make use of the catalog (this is available to you via the variable `myCatalog` in `main.cc`). You'll want to use the catalog to obtain all of the tables that are currently in the database, which will allow you to perform the required semantic checks.

2.3 Not Challenging Enough?

If you don't find all of this challenging enough, I am also shipping a minimal version of the project that contains very little code, other than the actual lexer and the parser. If you are up to it, use this minimal code as your starting point. If you use this version of the project, you have total freedom to design whatever data structures you want to store the parsed query, and you'll need to go into the parser itself and add code to build up those data structures.

2.4 My Code, Your Code, and the Honor Code

As usual, I am giving out my solution to A4 in this assignment. Simply stated, you are not allowed to distribute my code to **anyone** who is not taking the class right now… **ever**! Doing this constitutes a violation of the honor code. I'm particularly concerned about my code falling into the hands of students who might take the class in the future. Thus, a hard and fast prohibition on distributing my source code.

2.5 Testing And Grading

This assignment is different in that we are not using a set of automated unit tests to grade the assignment. Instead, when you are ready to submit, compile everything, and then (in the `Build` directory) fire up `bin/sqlUnitTest`. Copy and past the contents of `CreateTables.sql` into the command prompt, so that all of the TPC-H database tables are created. Then, one-by-one, copy and paste the seventeen test queries in `Test.sql` into the command prompt, and copy and paste the results of the semantic tests that you have implemented **into a text file.** Most of these queries have semantic errors, but some do not. Then, turn in a soft copy of your 17 results as a text file, along with your code as a zip file. Your score on this assignment is based on the number of those 17 queries on which you correctly perform a semantic check.

As usual: if you work with a partner, **only turn in one copy of your source and one hard copy of your results**. Otherwise, we'll possibly grade your submission twice, and end up getting quite annoyed.