Postgres Parser Internal Report

Hao Jin

1 Overview

The Postgres Parser component serves as the parsing module for Peloton. Prior to Postgres Parser, the system was running with a customized parser, which did not have a good enough coverage of the necessary types of queries and had some flaws. The creation of Postgres Parser was for the sake of providing reliable parsing for standard PostgreSQL queries so that the team can run a range of benchmarks on the system. The fact that Postgres Parser is not a real parser means that it is just a temporary solution, and if we want anything more than what PostgreSQL supports, we'll need a new parser module or an extra parser module.

2 How It Works

2.1 Pipeline

1. Get Postgres' parse tree by feeding the input query to *pg_query_parse* function
2. Pass the Postgres' parse tree, which is of type *List* *, to *ListTransform* function.
3. The *ListTransform* function will then traverse the list of parse nodes and call *NodeTransform* on each single one of them.
4. The *NodeTransform* function will then call the corresponding transform function for the type of parse node

2.2 Structure

Each transform function share the same basic logic: Traverse each field of the generated Postgres parse tree node and fill in the corresponding field in Peloton parse tree. If there's any necessary subsequent transform (e.g. transform of a sub-select in a select query), the transform function will call the corresponding transform function and wait for its result for filling in the corresponding field. Note that each transform function is only aware of its own sub-tree of the parse tree, thus knowing nothing about the rest parts in the parse tree, this design is to accommodate the nature of SQL parse trees and to provide re-usability of all transform functions. For example, a function responsible for transforming a constant value parse node can be used by the transform functions of both select queries and insert queries, and it does not, and should not be aware of which type of query it is working on.

2.3 Exception Handling and Memory Management

The current parser module only covers types of features that our system has encountered, there will definitely be features or types of queries or parse nodes not supported by the current version. The module is designed to throw NotImplementedException under such cases to let the user or developer be aware. On the other hand, when the input queries have some problems, the parser module should be able to identify that and throw exceptions.
With the above being said, one may notice that the exception can happen at any point of the transform, thus proper handling of exceptions is very important. There are two situations where the current version of parser module may encounter exceptions. One situation is that when the input query has some grammar problems, for this case the libpg_query library will automatically mark the ”error” field in the generated Postgres parse tree, our module will take it from there and throw and exception with the corresponding error message. The other situation is that when some transform function encounters a type of parse node or a feature that is not supported yet, and it will throw NotImplementedException. However, this could result in memory leaks if the exceptions are not properly handled.
The solution to this has 2 phases: For the first phase, we want to add memory management code along the calling paths so that no memory leak will happen when exceptions do occur in the future. For the second phase, we want to migrate to using smart pointers in the module so that we can utilize the underlying automatic memory management provided by the standard library. For the first phase, we should utilize the idea of defensive programming, every time we need to call some transform function, we should wrap a try-block around it and perform necessary memory management if necessary once we catch any exception. For the second phase, it requires a big change to the current code thus takes a longer time to complete, but the principle is quite easy which is to change the raw pointers to smart pointers.

3 Development Guidelines

3.1 Adding Support for Extra Node Types in Existing Transform Functions

One common type of addition to the parser module is that we notice that some queries introduce new types of child parse node to some kinds of parse trees that already have corresponding transform functions in our system, and the new type of child also has existing transform functions. So this is the easiest case, we can simply add a new case to the switch logic in the caller transform function to call the proper existing transform functions of the new types of child. If the child happens to encounter new types of child, follow this same guideline.
A good example for this is that we already support transform for TypeCast nodes, and we also have support for ResTarget Nodes. However we were not aware of the case where a TypeCast can be a

target. As we already have transform functions for both TypeCast and ResTarget, we can simply add this case to the transform function of ResTarget and let that function call TypeCast's transform function when some target is of TypeCast type.

3.2 Adding Support for New Type of Parse Node

For this case you should first find the corresponding definition for the parse node type in *third_party/libpg_query/src/postgres/include/nodes/parsenodes.h* and copy the necessary definitions to *src/include/parser/parsenodes.h*. Notice that if the new type of parse node depends on some other types you should also copy those.
Then you should add a new function in postgresparser.h/postgresparser.cpp to support this type of parse node. The general way of naming the transform functions is ¡Postgres parse node type name¿ + Transform. For example, the transform function for TypeCast is TypeCastTransform. The transform function usually takes in a pointer of the Postgres parse node type and returns a pointer to the corresponding type of Peloton parse node. For example, SelectTransform which is the transform function for SelectStmt nodes will return a pointer to SelectStatement which is Peloton's own type for select queries (Note SelectStmt's transform is not SelectStmtTransform because SelectTransform is shorter and easier to understand, so for all other kinds of statement types please get rid of the "Stmt" part in the name for transform functions).

3.3 Adding Support for New Type of Statements/Expressions

This is something similar to Adding support for new type of parse node, but notice that sometimes adding support for a new type of statement would also require additional statement types or re-organization of statement or expression systems. Please discuss with the whole team should this kind of needs pop up.

3.4 Style Guides for Transform Functions

The most important local variable for every transform functions is the final result of the transform function, which is a pointer to some kinds of Peloton's own parse node. To keep things consistent and easy to understand, please use "result" as the name of the local variable.
To make the interfaces clean and easy to understand, every transform function should explicitly take the corresponding parse node type as the input. For example, ConstTransform is the transform function for A_Const parse nodes, so it takes in a pointer to A_Const parse node as the argument.

4 Future Works

Some work that should be done recently is the refactorization of All CREATE-related statements. There should be a base class(CreateStatement) and various derived classes (CreateTableStatement, CreateIndexStatement, CreateSchemaStatement etc.).
This module should still be around for some good amount of time so that we can have reliable parsing and complete our own parse node system. The main framework is pretty much done for the time being, most of the future work on this module will be adding support for newly encountered types of queries and parse nodes.
On the other hand, Peloton should definitely need its own parsing module as there may be needs for parsing queries that do not belong to legal PostgreSQL queries, this should require a substantial amount of time and effort to complete. I think before we totally get rid of Postgres Parser module, we can have a 2-level design, the first level is still the Postgres Parser, while the second level can be a customized parser only for the new needs. In this way we can implement customized grammars in the second level while still have full support for normal SQL queries in the first level.