SqlParser readme

Table of Contents

Introduction

The pg_sqlparser can be used to parse the SQL dialect of PostgreSQL in a Java program. To keep as close as possible to the parser that is used by PostgreSQL itself, the actual PostgreSQL sources are used as basis for this parser.

The PostgreSQL sql parser consists of a scanner and a grammar.

Scanner

The scanner (also: "lexer") is the part of the parser that reads from an input stream and devides it into keywords, identifiers, characters etc. It filters out all comment and whitespace.

The source file of the scanner is src/backend/parser/scan.l. It is interpreted by Flex, which generates src/backend/parser/scan.c from it.

Grammar

The chuncks (keywords, identifiers, operators, characters, ...) that are returned by the scanner are put into context and interpreted by the grammar.

The source of the grammar is src/backend/parser/gram.y. It is passed to Bison, which creates src/backend/parser/gram.c from it.

Java v/s C

The scan.l and gram.y files are emphatically intended to be used in the programming language C. Well, that is fine for the database, but to create a maintainable system we prefer Java. Fortunately Bison can generate Java and for Flex we can use it's Java variant, called Jflex. So: "piece of cake" you might think. Well – not quite. The scan.l and gram.y files are littered with C constructs that the Java compiler does not understand, and C is very forgiving on type casts. So we created some programs to translate the C specifics to something that the Java compiler can deal with. For the grammar that generates a file that is syntactically correct Java, but it is so big that it runs into compiler limitations. So the generated Java source is processed so the Java compiler can deal with it (some exceptionally large tables are set aside as serialized Objects and the yyaction() method – which is basically one big switch statement – is split up into a few dozens of smaller methods).

Usage

The following code gives an example of how you can use the parser:

```
public static void main(String[] args) {
    String sql = "SELECT a, sum(b) AS total "
               + "FROM tbl "
               + "WHERE a != 'excl' "
               + "GROUP BY a";
    List<SqlParserErrorData> parserErrors = new ArrayList<>();
    try (Reader reader = new StringReader(sql)) {
        SqlParser parser = new SqlParser(reader
                                         , error -> parserErrors.add(error)
                                         );
        if (parser.parse()) {
            for (Node parsedStatement : parser.getResult()) {
                System.out.println("The parsed statement = "
                                   + parsedStatement
                                   );
                System.out.println(
                        "The statement in XML format looks like:\n"
                        + ParserUtil.stmtToXml(parsedStatement)
                        );
            }
        } else {
            for (SqlParserErrorData err : parserErrors) {
                System.out.println("Parser error: " + err);
            }
        }
    } catch (IOException e) {
        System.out.println(e);
    }
}
```

will show the parsed and re-generated sql statement and the xml representation of the statement.

Project structure

The SqlParser project contains four modules:

Module	Description
parser_enums	Contains enums that are used in the grammar-to-java module to help transforming the gram.y file, and that are used in the resulting parser.
grammar-to-java	Source code for the maven pugins that are used in the parser module to perform the necessary conversions in the parser generation process.
parser	The actual parser, which is the objective of the project.
parser-test-gui	Holds a gui that can be very useful in debugging.

Module parser-enums

The parser-enums module contains holds enums that are used by the parser. Every enum holds a public static final String REPLACEMENT_REGEXP_PART, that is used in com.splendiddata.sqlparser.grammartojava.GrammarConverter to make a proper Java enum value of the literals that are used in the gram.y file.

Module grammar-to-java

In the grammar-to-java module the following classes in package com.splendiddata.sqlparser.grammartojava define Maven plugins for the parser module:

Class	Description
LexConverter	Reads the scan.l file and converts it to a version that generates Java code instead of C when passed to Jflex.
GrammarConverter	Converts the gram.y file to a version from which Bison can generate a syntactically almost correct Java source file.
JavaParserConverter	Reads the Java source file that is generated by Bison from the converted gram.y file. It converts some very big arrays into serialized objects that can be loaded at runtime by the parser. And it splits up the yyaction() methode in chuncks that can be processed by the Java compiler. This because the Java compiler has a hard limit of 64kB generated object code per method.
JaxbIndexCreator	Every structure that is generated by the parser is annotated with javax.xml.bind.annotation annotations. The advantage of that is that the structure of objects that are generated by the parser can be visualised nicely as an xml structure. This can make life a lot easier while debugging an application that uses the parser. But to be able to make fully use of the annotations using JAXB, jaxb.index files need to be generated for every package that holds annotated classes. The JaxbIndexCreator does exactly that.

LexConverter

The LexConverter class reads the scan.l file and replaces some C peculiarities by their Java counterparts. Pointers are converted to Java references (my_struct* becomes my_struct, my_struct->field becomes my_struct.field etc.). Some sections in the scan.l file need special treatement. Therefore the LexRuleSpecial class is used.

The LexConverter also generates the file header, so the generated file can be processed propery by JFlex.

GrammarConverter

If you like regular expressions, this class will make you ecstatic. “Normal” people might need a bucket.

In principle the GrammarConverter does the same as the LexConverter:

convert C constructs into Java, but on a much larger scale.

It makes use of the GrammarRuleSpecial enum for special processing of specific sections in the gram.y file. The name of the enum value exactly matches the name of the section in the gram.y file.

The converted gram.y file can be passed to Bison to generate a syntactically correct Java source file.

JavaParserConverter

In the Java source file that is generated by Bison from the converted gram.y file, a few challenges remain:

C doesn't mind casting any type to any other type – you are the boss. So the grammar developers did not always use the correct type casts. But Java has a different opinion on that. So some repair work is still to be done by the JavaParserConverter.

And the guys that invented Java could not imagine that any developer would ever need a single method that would generate more than 64kB of bytecode (why would you size a teacup to contain an ocean?). But the generated Java file has two methods that go well beyond that limit: the yyaction() method and the static initializer.

The static initializer is so big because it initialises a couple of tables that are well over 64kB themselves. We resolved that problem by serializing the tables into files that are loaded at runtime by the parser.

And the yyaction() method is basically a big well-ordered switch statement, which could be split up easily into a bunch of smaller switch statements. This even results in a small performance gain as many cases don't need to be checked any more.

JaxbIndexCreator

Uses a Java parser to find every class that is annotated with a javax.xml.bind.annotation.* annotation and to make a jaxb.index entry for it in it’s package. Beware of the fact that JAXB is a bit over-sensitive. If any class is not exactly right according to JAXB, then all XML processing of all classes will fail. So do make sure that every class with a javax.xml.bind.annotation.* does have a no-argument constructor and that JAXB understands the return type of every getter (unless it is annotated with javax.xml.bind.annotation.XmlTransient).

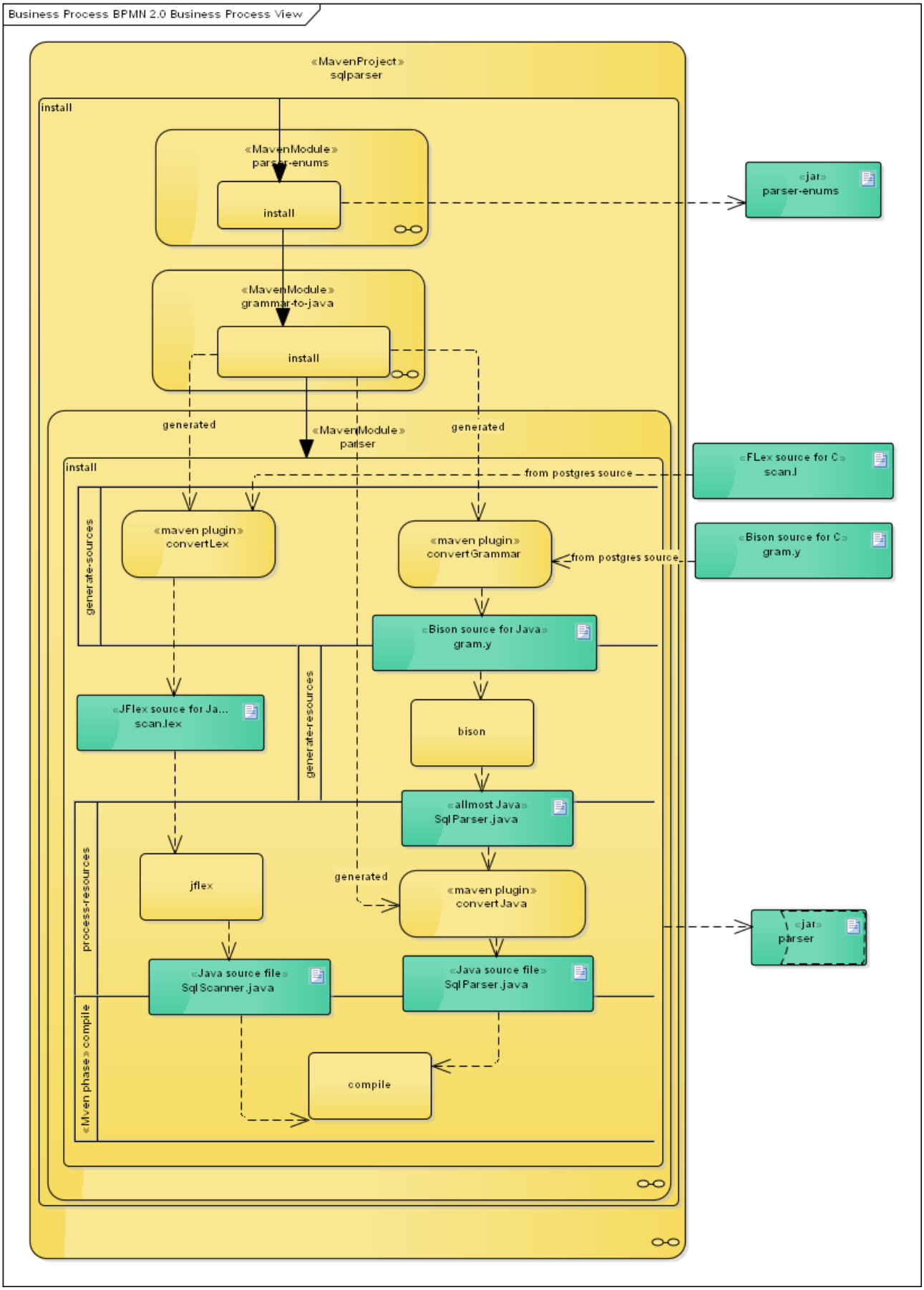
Module parser

The Maven build process is a bit complex.

It consists of:

- Preprocessing the src/main/scanner/postgres/src/backend/parser/scan.l file so that JFlex can make Java of it. The converted file will be placed in target/generated-sources/converted.scanner/scan.lex.
- Using JFlex, generate target/generated-sources/jflex/com/splendiddata/sqlparser/SqlScanner.java from target/generated-sources/converted.scanner/scan.lex.
- Preprocessing src/main/grammar/postgres/src/backend/parser/gram.y to target/generated-sources/converted.grammar/gram.y so that Bison can make "Java" of it.
- Generating target/generated-sources/generated.java/com/splendiddata/sqlparser/SqlParser.java from the target/generated-sources/converted.grammar/gram.y file using Bison.
- Converting target/generated-sources/generated.java/com/splendiddata/sqlparser/SqlParser.java to target/generated-sources/converted.java/com/splendiddata/sqlparser/PgSqlParser.java and target/generated-sources/converted.java/com/splendiddata/sqlparser/ScanKeyword.java and extract a couple of large tables to target/arrays/…
- Creating a jaxb.index for every package that contains classes with JAXB annotations.
- Compiling the Java source files.

The following picture shows the process (including the parser-enums and grammar-to-java modules):


Business Process BPMN 2.0 Business Process View
«MavenProject»
sqlparser
install
«MavenModule»
parser-enums
install
«jar»
parser-enums
«MavenModule»
grammar-to-java
install
generated
«MavenModule»
parser
generated
install
from postgres source
«FLex source for C»
scan.l
generate-sources
«maven plugin»
convertLex
«maven plugin»
convertGrammar
from postgres source
«Bison source for C»
gram.y
«Bison source for Java»
gram.y
«JFlex source for Ja...
scan.lex
generate-resources
bison
«allmost Java»
SqlParser.java
process-resources
jflex
generated
«maven plugin»
convertJava
«jar»
parser
«Java source file»
SqlScanner.java
«Java source file»
SqlParser.java
«Mven phase» compile
compile

Package com.splendiddata.sqlparser.structure

The scaner: com.splendiddata.sqlparser.SqlParser will return structures that are defined in the com.splendiddata.sqlparser.structure package.

As the SqlParser was originally intended to be used in C, no getters and setters are used for the properties. So practically all properties are public.

All structures implement clone().

All structures have a copy constructor.

All structures implement toString(). The idea behind that is that the original query can be re-generated from the parsed object structure. Well – the generated query must be functionally the same. Some minor differences might occur, as long as the database will treat them equally.

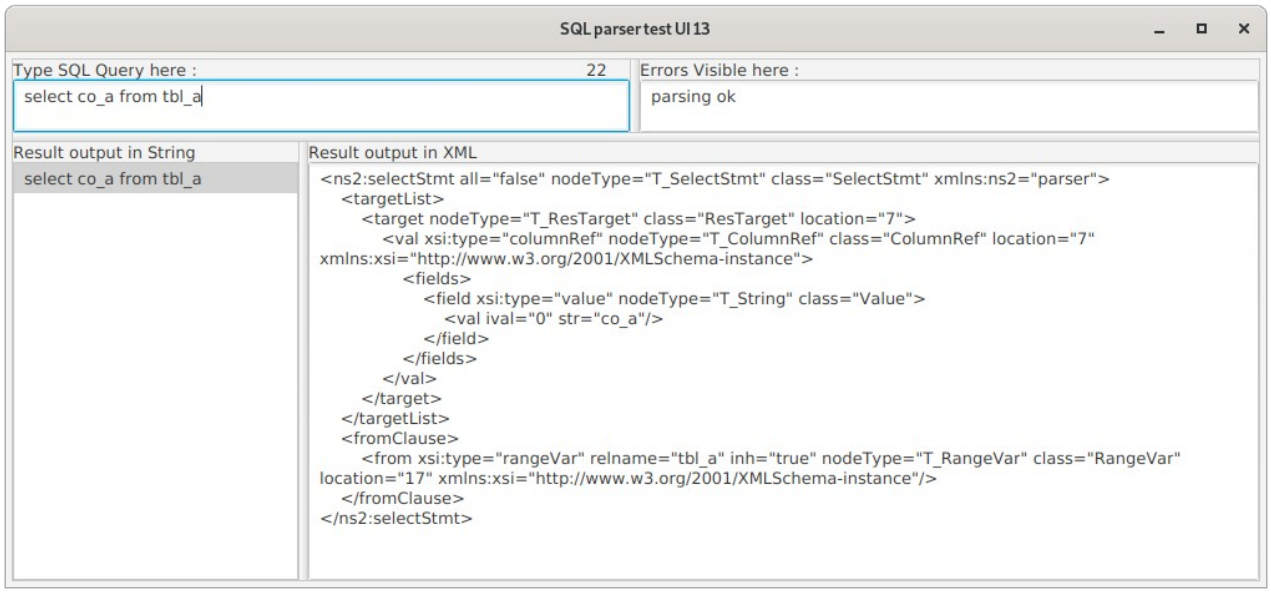
No maven plugin for Bison.

For Bison no maven plugin is available. So Bison must be installed in the operating system.

(on RedHat-like systems: “yum install bison”)|.

Module parser-test-gui

The module parser-test-gui holds, as it says, a test gui that looks like this:



It consists of four text areas.

In the top left-hand text area, you can type a sql statement.

The top right-hand area will show an error message if applicable.

The bottom left-hand area will show the re-generated statement. The sql statement from the top left-hand area is parsed to an object structure, and then toString() is used on the object structure to generate the statement in the bottom left-hand area. The statement here should be functionally the same as the one in the top left-hand area. If the parser-test-gui is compiled 'withFormatter', then you can right-click on a statement in the bottom right-hand area to format it.

In the bottom right-hand text area, the object structure that is generated by the parser as a result of parsing the text in the top left-hand area, is visualised in an XML structure.

Compilation

The parser-test-gui Maven module has two profiles: 'noFormatter' and 'withFormatter'. Profile 'withFormatter' depends on com.splendiddata.pgcode_formatter:pgcode_formatter_main, which as a test dependency on com.splendiddata.pg_sqlparser:parser. So, if you want to use toe formatter in the bottom left-hand text area, it is best to first do a compile the pg_sqlparser with the default profile (= 'noFormatter'), then compile

the pgcode_formatter and then compile the test gui again using profile ‘withFormatter’.

Execution

The parser-test-gui makes use of javafx. Since Java9 it is a bit of a challenge to make that execute properly. “mvn exec:java” works fine.

New version of PosgreSQL

A new version of PostgreSQL comes with a new version of its parser. So the SqlParser needs to be adapted.

To adapt this parser to the official PostgreSQL parser, please roughly follow the following steps:

Create a new version of the SqlParser

Branch off the current version in Git

Alter the pom files to reflect the new version number. The version number is kept in line with the PostgreSQL major version.

Update version information in the ParserUtil

In `com.splendiddata.sqlparser.ParserUtil`, methods `getParserVersion()` and `getForPostgreSQLVersion()` need to be changed to reflect proper version information for the version that we are going to implement.

Download the PostgreSQL source code

http://www.postgresql.org/ will lead you to the newest source version.

Synchronise com.splendiddata.sqlparser.Keyword

com.splendiddata.sqlparser.Keyword must be in synch with src/include/parser/kwlist.h in the PostgreSQL source.

Copy the scan.l and gram.y file

Copy the src/backend/parser/scan.l file from the PostgreSQL source to parser/src/main/scanner/postgres/src/backend/parser/scan.l

Workaround (JFlex known issue):

> According to JFlex manual, the syntax of the ”lexical rules” is
> Rule ::= [StateList] [’^’] RegExp [LookAhead] Action
> | [StateList] ’<<EOF>>’ Action
> | StateGroup
> Action ::= ’{’ JavaCode ’}’ | ’|’
> However, if the action for an <<EOF>> rule is a '|', then the generator fails and gives the error:
> <<EOF>> must be followed by an action.
> So, in the file scan.l, the examples like:
> <xusend><<EOF>> |
> <xusend>{other} |
> <xusend>{xustop1} { ... }
>
> should be changed as follows (<<EOF>> should be directly followed by an explicit action, not

```
by a pipe '| '):
<xusend>{other} |
<xusend>{xustop1} |
<xusend><<EOF>> { ... }
```

For the time being, this workaround can be used. Hopefully this issue will be solved in a next release of JFlex, otherwise we have to adapt the parser.

Copy the src/backend/parser/gram.y file from the PostgreSQL source to parser/src/main/grammar/postgres/src/backend/parser/gram.y

Adapt the converter programs, enums and structure classes

Build, find the errors, resolve them (get your inspiration for enum and strucure classes from the PostgreSQL source) and build again. Nearly all files that are derived from C structures have a reference to the C source file on which they are inspired. So in most cases you will be able to spot differences that are to be resolved. Be creatie. Good luck.

Adapt test files

Under parser/src/test/resources, a lot of test sql files can be found. These files are all parsed, and then each statement in them is rendered from the structure that comes out of the parser, and then parsed and rendered again. The second rendering must result in the same string as the first rendering. The first rendering should functionally result in the same statement as the original one, but will not be an exact match because of non-functional differences in whitespace, comments and casing. So unfortunately this must be checked manually. Find your inspiration on which files to check manually in the PostgreSQL release notes.

Files under parser/src/test/resources/commands are created by us. Feel free to extend them in any way you like. Please try to reflect as many situations as you can in these files. Pay special attention to the differences mentioned in PostgreSQL's release notes.

Files under /parser/src/test/resources/postgres are literally copied from the src/test/resources/postgres/test/regress/sql directory in the PostgreSQL source. But because some statements in there are intended to fail, and we don't have a possibility to distinguish them (yet?), some of the test cases are commented out. These cases are documented in the files:

Altered test files are provided with a header reading:

```
/*
 * This file has been altered by SplendidData.
 * It is only used for syntax checking, not for the testing of a commandline paser.
 * So input for the copy statements is removed.
```

```
 * The deactivated lines are marked by: -- Deactivated for SplendidDataTest:
 */
```

And the commented out lines start with:

```
-- Deactivated for SplendidDataTest:
```

Please do keep PostgreSQL's regression test files as close to the original ones as possible.