Runtime-Extensible Parsers

Hannes Mühleisen and Mark Raasveldt
DuckDB Labs
Amsterdam, The Netherlands
firstname@duckdblabs.com

ABSTRACT

Despite their central role in processing queries, parsers have not received any noticeable attention in the data systems space. State-of-the art systems are content with ancient old parser generators. These generators create monolithic, inflexible and unforgiving parsers that hinder innovation in query languages and frustrate users. We argue that parsers should be rewritten using modern abstractions like Parser Expression Grammars (PEG), which allow dynamic changes to the accepted query syntax and better error recovery. In this paper, we discuss how parsers could be re-designed using PEG, and validate our recommendations using experiments for both effectiveness and efficiency.

1 INTRODUCTION

The parser is the DBMS component that is responsible for turning a query in string format into an internal representation which is usually tree-shaped. The parser defines which queries are going to be accepted at all. Every single SQL query starts its journey in a parser. Despite its prominent position in the stack, very little research has been published on parsing queries for data management systems. There seems to have been very little movement on the topic in the past decades and their implementations are largely stuck in sixty year old abstractions and technologies.

The constant growth of the SQL specification with niche features (e.g. support for graph queries in SQL/PGQ or XML support) as well as the desire to support alternative query notations like dplyr [18], piped SQL [15], PRQL[1] or SaneQL [13] makes monolithic parsers less and less practical: In their traditional design, parser construction is a *compile-time* activity where enormous grammar files are translated into state machine transition lookup tables which are then baked in a system binary. Having those *always* be present in the parser might be wasteful especially for size-conscious binary distributions like WebAssembly (Wasm).

Many if not most SQL systems use a static parser created using a YACC-style [10] parser toolkit: We are able to easily confirm this for open-source systems like PostgreSQL and MySQL/MariaDB. From analyzing their binaries' symbol names, we also found indications that Oracle, SQL Server and IBM Db2 use YACC. Internally, YACC and its slightly more recent variant GNU Bison as well as the "Lemon" parser generator used by SQLite all use a "single look-ahead left-to-right rightmost derivation" LALR(1) parser generator. This generator translates a formal context-free set of grammar rules

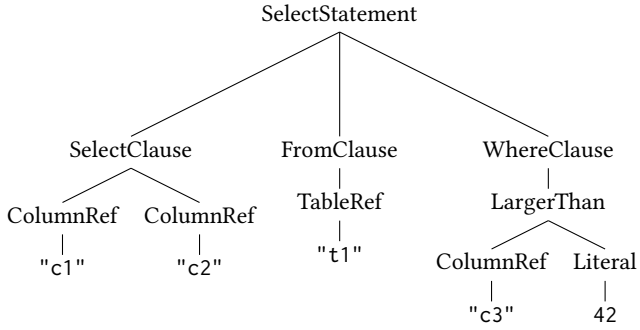

Figure 1: Example AST

in Extended Backus-Naur Form (EBNF) to a parser state machine. LALR parsers [4] are a more space-efficient specialization of LR(k) parsers as first described by Knuth [8]. But in effect, the most advanced SQL systems of 2024 use parser technology from the 1960s. Given that the rest of data management systems have been greatly overhauled since this should raise the question why the parser did not receive any serious engineering attention.

Database systems are moving towards becoming *ecosystems* instead of pre-built monoliths. Much of the innovation in the PostgreSQL, SQLite, and DuckDB communities now comes from *extensions* [7], which are shared libraries that are loaded into the database system at run-time to extend the database system with features like vector similarity search, geospatial support, file systems, or graph processing. Bundling all those features upfront would be difficult due to additional binary size, external dependencies. In addition, they are often maintained independently by their communities. Thus far, at least in part due to the ubiquity of YACC-style parsers, those community extensions have been restricted from extending syntax. While this is also true in other ecosystems like Python, the design of SQL with its heavy focus on syntax and not function calls makes the extensions second-class citizens that have to somehow work around the restrictions by the original parser, e.g. by embedding custom expressions in strings.

We propose to *re-think data management system parser design* to create modern, *extensible* parsers, which allow a dynamic configuration of the accepted syntax *at run-time*, for example to allow syntax extensions, new statements, or to add entirely new query languages. This would allow to break up the monolithic grammars currently in use and enable more creativity and flexibility in what syntax a data management system can accept, both for industrial and research use. Extensible parsers allow for new grammar features to be easily integrated and tested, and can also help bridge the gap between different SQL dialects by adding support for the dialect of one system to the parser of another. Conversely, it might also be desirable in some use cases to *restrict* the acceptable grammar, e.g.

[1] https://prql-lang.org/

 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands

to restrict the complexity of queries, or to enforce strict compliance with the SQL standard.

Modernizing parser infrastructure also has additional benefits: One of the most-reported support issues with data management systems are unhelpful syntax errors. Some systems go to great lengths to try to provide a meaningful error message, e.g. "this column does not exist, did you mean ...", but this is typically limited to resolving identifiers following the actual parsing. YACC-style parsers exhibit "all-or-nothing" behavior, the *entire* query or set of queries either is accepted entirely or not at all. This is why queries with actual syntactical errors (e.g. "SELEXT" instead of "SELECT" are usually harshly rejected by a DBMS. MySQL for example is notorious for its unhelpful error messages:

"You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'SELEXT' at line 1."

In this paper, we make the following contributions:

- We describe the current state of parsers in data management systems
- We propose to switch to run-time generated parsers
- We present an initial design of a runtime-extensible SQL parser
- We perform proof-of-concept functionality and efficiency experiments using a prototype SQL parser

The rest of this paper is structured as follows: Section 2 introduces parser technology background. Section 3 describes our proposal to use Parsing Expression Grammars (PEG) to build an extensible SQL parser. Section 4 contains a description of our prototype SQL parser and several experimental results. Finally, Section 5 concludes this paper.

2 PARSER BACKGROUND

Parsing is the first step in the "textbook" database system design. An arbitrary query arrives from the user, e.g. via API call or client protocol. The query is then parsed into an abstract syntax tree. Once parsed, we can assume that the query is syntactically correct. Most dreaded syntax errors originate from this step.

In the subsequent *binding* step, the symbolic identifiers in the syntax tree are bound to the available types, functions, schema, table, and column names etc. At this point the query is semantically correct and execution failures are typically issues in the actual table data, e.g. casting errors. Next various planning, optimization and execution steps follow. In the following, we will focus exclusively on the transformation between input SQL string and abstract syntax tree, i.e. the actual parsing.

Parsing of a formal language, be it SQL or Python, consists of two parts: *Lexical Analysis* (also called the "lexing" or "tokenization") and *Syntactic Analysis* (also confusingly called the "Parser" itself) [1].

Lexical Analysis splits an opaque input string into a list of still-opaque "tokens". Often, regular expressions are used to define what are considered valid tokens. For example, the SQL query `"SELECT c1, c2 FROM t1 WHERE c3 > 42"` could be tokenized to the list `"SELECT", "c1", "c2", "FROM", "t1", "WHERE", "c3", ">", "42"` with `"` denoting token boundaries. As can be seen, this lexer removes the white space present in the original queries but it can also be used to e.g. strip comments. There is no analysis of the parsed tokens, for example, the lexer would happily tokenize `"SELEXT 42"` or `"FROM t1 SELECT c1, c2"`, both of which are not usually allowed in SQL[2].

Note that there are types of parsers that do not have a separate lexical analysis step, in that case, syntax analysis is performed on a character-by-character level. Furthermore, lexical analysis does not actually need to materialize the entire token list for a given input but can expose an iterator-style interface to return tokens on a one-by-one basis.

Syntactic Analysis interprets the list of tokens according to a defined formal grammar and decides whether the list is acceptable syntax. The result of parsing typically is a hierarchical abstract syntax tree (AST). The example query above could be transformed into the exemplary AST in Figure 1 given a hypothetical SQL grammar:

As an example for a SQL grammar, Figure 2 is a simplified snippet of the 20.000 lines (!) of Bison grammar in Postgres that shows the crucial `simple_select` rule that interprets `SELECT` queries:

```
from_clause:
 FROM from_list

from_list:
 table_ref | from_list ',' table_ref

where_clause:
 WHERE a_expr

group_clause:
 GROUP_P BY set_quantifier group_by_list

simple_select:
 SELECT opt_all_clause opt_target_list
 into_clause from_clause where_clause
 group_clause having_clause window_clause
```

Figure 2: Postgres YACC Grammar Snippet

Given the token list and the grammar, there are now two main ways to decide whether the input matches the grammar: *Top-down* and *Bottom-up* [1]. In top-down parsing, the formal grammar rules are expanded top-down from the starting symbol to eventually match the entire input token list. In the example above, the starting symbol would be `simple_select`. In contrast, bottom-up parsers will start with the list of input tokens and try to combine ("reduce") those tokens into higher-level symbols to eventually arrive at the start symbol. If this succeeds, the input is accepted. In general, bottom-up parser were long considered to be more time and space efficient than top-down parsers [1]. YACC's LALR(1) parser is an example of a bottom-up parser [4].

[2]DuckDB actually allows the `FROM`-first syntax, see https://duckdb.org/docs/sql/query_syntax/from

2.1 YACC Internals

In YACC, tokens from the lexical analysis step are read and pushed ("shifted") onto a stack. Once the tokens on the stack match the components of a grammar rule (also called a "grouping"), the tokens are "reduced" on the stack to the left hand side of the grammar rule. This is efficiently implemented by generating a (possibly large) look-up table during grammar compilation for each possible combination of tokens and groupings on the stack [3]. At each step, the parser can use the current stack as an index into the look-up table and retrieve the appropriate action (e.g. "read next token", "apply reduction xyz", or "produce error"). When a token is read from the input, it is not immediately shifted onto the stack to allow the single look-ahead to disambiguate if necessary. On reduction, a so-called "action" can be invoked. Actions are actual program code (with some placeholders) that are inserted into the generated parser code and can perform arbitrary transformations of the reduced tokens, e.g. creating parse tree objects. Postgres' parser for example makes heavy use of reduce actions to create the abstract syntax tree for the query (`PGNode`).

If the parser cannot decide whether tokens should be reduced or additional tokens should be read from the input, the grammar is undecidable and a so-called "shift-reduce conflict" is thrown at compile time. Similarly, if two rules could be applied, a "reduce-reduce conflict" is present [3]. When adding new syntax, avoiding shift-reduce conflicts is a common nuisance that increases the complexity and effort to extend the parser. In certain cases, it is even impossible to add new desired syntax because there is no way to do so without triggering conflicts.

In the case of Postgres, YACC is only barely able to parse its SQL dialect cleanly. There are already several hacks in the lexical analyzer because in some cases its SQL grammar requires more than a single token look-ahead to be conflict-free, for example due to the multi-token keywords like `ORDER BY`. This introduces complications due to possible comments in between tokens, e.g. `ORDER /* comment */ BY`. To work around limitations of the single-token look-ahead the lexical analyzer these multi-token keywords are explicitly detected and replaced by a single token.

2.2 Parsing Expression Grammar

Parsing Expression Grammar (PEG) parsers represent a more modern approach to parsing [6]. PEG parsers are top-down parsers that effectively generate a recursive-descent style parser from a grammar. Through the "packrat" memoization technique PEG parsers exhibit linear time complexity in parsing at the expense of a grammar-dependent amount of extra memory. The biggest difference from a grammar author perspective is the choice operator where multiple syntax options can be matched. In LALR parsers options with similar syntax can create ambiguity and reduce conflicts. In PEG parsers the *first* matching option is always selected. PEG parsers cannot be ambiguous by design [6].

As their name suggests, parsing expression grammar consists of a set of *parsing expressions*. Expressions can contain references to other rules, or literal token references, both as actual strings or character classes similar to regular expressions. Expressions can be combined through sequences, quantifiers, optionals, groupings and both positive and negative look-ahead. Each expression can either match or not, but it is required to consume a part of the input if it matches. Expressions are able to look ahead and consider the remaining input but are not required to consume it. Lexical analysis is typically part of the PEG parser itself, which removes the need for a separate step.

One big advantage is that PEG parsers *do not require a compilation step* where the grammar is converted to for example a finite state automaton based on lookup tables. PEG can be executed directly on the input with minimal grammar transformation, making it feasible to re-create a parser at runtime. PEG parsers are gaining popularity, for example, the Python programming language has recently switched to a PEG parser [17].

Another big advantage of PEG parsers is *error handling*: [11] describes a practical technique where parser rules are annotated with "recovery" actions, which can 1) show more than a single error and 2) annotate errors with a more meaningful error message.

A possible disadvantage of memoized packrat parsing is the memory required for memoization: The amount required is *proportional to the input size*, not the stack size. Of course, memory limitations have relaxed significantly since the invention of LALR parsers sixty years ago and queries typically are not "Big Data" themselves.

```
Expression <- ColumnReference / StringLiteral /
    OperatorExpression
OperatorExpression <- Expression Operator Expression
ColumnReference <- '"' [^"]* '"'
StringLiteral <- '\'' [^\']* '\''
Operator <- ('AND' / 'OR' / '=' / '<' / '>')
```

Figure 3: "Illegal" Left-Recursive PEG Rules

Canonical PEGs have one systemic restriction: Grammars cannot be *left-recursive* [6]. Consider the example in Figure 3: We wish to define a simple expression grammar, where expressions can either be column references enclosed in double quotes (e.g. `"my_column"`), a string literal in single quotes (e.g. `'my_literal'`) or a binary comparison of two other expressions. While this seems reasonable, the recursive descent that occurs while parsing this will lead to infinite recursion, because an `Expression` can contain another one without consuming input. This will go on ad nauseam (or until the stack overflows).

```
OperatorExpression <- Expression (Operator Expression)*
Expression <- ColumnReference / StringLiteral
```

Figure 4: Non Left-Recursive PEG Rules

This issue can canonically be fixed by adding another rule with an optional repeat clause like in Figure 4. Now, `OperatorExpression` is not just another `Expression` but instead a separate, higher-level concept just to avoid left recursion. However, this becomes challenging for some SQL constructs like `x BETWEEN y AND z`, where the first part `x` can in principle be any other kind of expression, including another `BETWEEN` expression. This necessitates additional grammar rules where some expressions are expressed outright in some situations, and others require additional tokens in the input to disambiguate (in SQL, usually parentheses).

This issue can be addressed by limiting the stack depth of the generated recursive call to the number of detected operator tokens. In essence, if the maximum stack depth has been reached, the generated parser will no longer choose the recursive rule to instantiate an expression, but will instead force instantiation of the other alternatives if present. There exists an efficient implementation of this technique by inverting the call stack and with memoization [9, 12].

In the following, we will describe how PEG can be used to build a runtime-extensible SQL parser.

3 EXTENDING SQL

SQL has long been extensible at run-time – in a way: Most systems do for example not hard-code the list of available scalar, aggregate, window, or table-producing *functions* in the parser. Instead, the available functions are dynamically resolved in the binding step, with the parser only returning an opaque function name. Function definitions can either come from an internal catalog or be supplied by the user. For example, Postgres supports the `CREATE FUNCTION` syntax which allows creating a function, e.g. written in the internal PL/pgSQL language or as a symbol in an external binary. Once defined, the function can be used in queries.

Due to the idiosyncrasies of SQL this approach can be challenging already. For example, whether a function is a scalar or an aggregate function dictates whether a query contains an aggregation or not, that major decision hence needs to be deferred until after binding. With generic function binding, the query `SELECT f() FROM t1` contains a top-level aggregate if `f` is an aggregate.

Similar to functions, some SQL systems allow for the definition of additional user-defined `data types`. For example, Postgres has the `CREATE TYPE` syntax for this. Again, the available data types are not hard-coded in the parser, but instead resolved dynamically in the binding step. Some extensions for Postgres, for example "pgvector", also add additional types implicitly. One restriction with user-defined types is that their literals need to be expressed *within* SQL string literals with `'...'`.

In principle, any new functionality could be added as table-producing functions. For example, we could add support for the Pipelined Relational Query Language (PRQL) like so:

```
SELECT * FROM
  execute_prql('from invoices aggregate (count this)');
```

We can see that the foreign query language is embedded in a normal, "parseable" SQL query as a literal string. The `execute_prql` function can take this string, execute as it wishes, and return an intermediate table. However, there are severe usability issues with this approach: There is little to no help with syntax highlighting or auto-completion in an editor, and quotes will possibly have to be escaped. It is also impossible to combine new and existing query language features without duplicating large parts of parsing and execution infrastructure in the table-producing function.

Because of the rigidity of SQL parsers, foreign language integrations via strings like in the example above are quite common. Standardization proponents will argue that making query languages extensible will lead to even more substantial deviations from the standard. This is of course a valid concern and the standardization of SQL has been a general success story. However, adding syntax by passing strings to "magic" functions merely moves the incompatibility to another layer, namely the function resolution and execution. It is likely preferable to fail early in those cases. In addition, there is a lot of syntax that is standardized (e.g. SQL/PGQ) that a database system might not want to enable by default but instead enable through an extension that modifies the parser to accept the syntax. A clear downside of extending syntax is when queries pass through generic SQL middleware which is going to be unaware of any non-standard extensions.

3.1 Adding New Statements

Let's assume we would want to add a new top-level `UNPIVOT` statement to turn columns into rows to a SQL dialect. `UNPIVOT` should work on the same level as e.g. `SELECT`, for example to unpivot a table `t1` on a specific list of columns or all columns (`*`), we would like to be able to write:

```
UNPIVOT t1 ON (c1, c2, c3);
UNPIVOT t1 ON (*);
```

It is clear that we would have to somehow modify the parser to allow this new syntax. However, when using a YACC parser, this would require modifying the grammar, re-running the parser generator, hoping for the absence of shift-reduce conflicts, and then recompiling the actual database system. However, this is not practical at run-time which is when extensions are loaded, ideally within milliseconds.

To partially address this issue, DuckDB [14] supports a workaround for this problem so that extensions can add top-level statements, so-called "fallback parsers". Fallback parsers are additional parsing functions that can be registered by extensions. Whenever the "normal" parser would reject a statement, it invokes the fallback parser in turn. The fallback parsers can then choose to accept a particular statement and return a simple query plan. This method is for example used by the MotherDuck extension [2].

However, it is impossible to accept statements allowed by fallback parsers in (for example) subqueries that are being read by the main parser without *duplicating* the entire original parser. Even if the original parser were duplicated, consider the case where *multiple* extensions wish to register different syntax extensions, since they are likely unaware of each other mixing statements is impossible.

It might be possible to ship the grammar files together with a modified YACC parser generator (as of yet non-existent "libbison") as part of the database runtime. Then, extensions could append new grammar rules to the grammar, and invoke the parser generator again to create the parser state transition look-up tables. This approach could work for multiple extensions registering multiple syntax extensions, and it would even be possible to mix statements defined by different extensions. This process would be able to check the added syntax by rejecting any additions that create conflicts. However, the design of YACC is *heavily* based on the assumption that the parser generator creates e.g. C source files that are then run through an actual compiler to produce the actual parser. Again, in theory, since some DBMSs already contain full-blown compilers, they could use those compilers to re-generate their parsers. It is likely more effective to switch to a modern PEG parser which can be evaluated without compile-time grammar pre-processing.

3.2 Extending Existing Statements

Let's now assume we would want to modify the SELECT syntax to add support for SQL/PGQ graph matching patterns [5]. Figure 5 is an example query in SQL/PGQ that finds the university name and year for all students called Bob:

```
SELECT study.classYear, study.name
FROM GRAPH_TABLE (pg,
MATCH (a:Person WHERE a.firstName ='Bob')-
    [s:studyAt]->(u:University)
COLUMNS (s.classYear, u.name)) study;
```

Figure 5: SQL/PGQ Example

We can see that this new syntax adds the GRAPH_TABLE clause and the pattern matching domain-specific language (DSL) within. To add support for this syntax to a SQL parser at runtime, we need to modify the grammar for the SELECT statement itself. This is fairly straightforward when using a PEG. We replace the rule that describes the FROM clause to also accept a sub-grammar starting at the GRAPH_TABLE keyword following by parentheses. Because the parser does not need to generate a state machine, we are immediately able to accept the new syntax.

4 PROTOTYPE & EXPERIMENTS

To perform experiments on parser extensibility, we have implemented an – admittedly simplistic – experimental prototype PEG parser for enough of SQL to parse *all* the TPC-H and TPC-DS queries. This grammar is compatible with the cpp-peglib single-header C++17 PEG execution engine[3].

cpp-peglib uses a slightly different grammar syntax, where / is used to denote choices. The symbol ? shows an optional element, and * defines arbitrary repetition. The special rules Parens() and List() are grammar macros that simplify the grammar for common elements. The special %whitespace rule is used to describe tokenization.

Below is an abridged version of our experimental SQL grammar, with the Expression and Identifier syntax parsing rules omitted for brevity:

All experiments were run on a 2021 MacBook Pro with the M1 Max CPU and 64 GB of RAM. The experimental grammar and the code for experiments are available on GitHub[4].

Loading the base grammar from its text representation into the cpp-libpeg grammar dictionary with symbolic rule representations takes 3 ms. In case that delay should become an issue, the library also allows to define rules programmatically instead of as strings. It would be straightforward to pre-compile the grammar file into source code for compilation, YACC-style. While somewhat counter-intuitive, it would reduce the time required to initialize the initial, unmodified parser. This difference matters for some applications of e.g. DuckDB where the database instance only lives for a few short milliseconds.

For the actual parsing, YACC parses TPC-H Query 1 in ca. 0.03 ms, where cpp-libpeg takes ca. 0.3 ms, a ca. 10× increase. To further stress parsing performance, we repeated all TPC-H and TPC-DS queries six times to create a 36,840 line SQL script weighing in at ca. 1 MB. Note that a recent study has found that the 99-percentile of read queries in the Amazon Redshift cloud data warehouse are smaller than 16.5 kB [16].

```
Statements <- SingleStmt (';' SingleStmt )* ';'*
SingleStmt <- SelectStmt
SelectStmt <- SimpleSelect (SetopClause SimpleSelect)*
SetopClause <-
    ('UNION' / 'EXCEPT' / 'INTERSECT') 'ALL'?
SimpleSelect <- WithClause? SelectClause FromClause?
    WhereClause? GroupByClause? HavingClause?
    OrderByClause? LimitClause?
WithStatement <- Identifier 'AS' SubqueryReference
WithClause <- 'WITH' List(WithStatement)
SelectClause <- 'SELECT' ('*' / List(AliasExpression))
ColumnsAlias <- Parens(List(Identifier))
TableReference <-
   (SubqueryReference 'AS'? Identifier ColumnsAlias?) /
    (Identifier ('AS'? Identifier)?)
ExplicitJoin <- ('LEFT' / 'FULL')? 'OUTER'?
    'JOIN' TableReference 'ON' Expression
FromClause <- 'FROM' TableReference
    ((',' TableReference) / ExplicitJoin)*
WhereClause <- 'WHERE' Expression
GroupByClause <- 'GROUP' 'BY' List(Expression)
HavingClause <- 'HAVING' Expression
SubqueryReference <- Parens(SelectStmt)
OrderByExpression <- Expression ('DESC' / 'ASC')?
    ('NULLS' 'FIRST' / 'LAST')?
OrderByClause <- 'ORDER' 'BY' List(OrderByExpression)
LimitClause <- 'LIMIT' NumberLiteral
AliasExpression <- Expression ('AS'? Identifier)?
%whitespace <- [ \t\n\r]*
List(D) <- D (',' D)*
Parens(D) <- '(' D ')'
```

Figure 6: Simple SQL grammar for PEG

Postgres takes on average 24 ms to parse this file using YACC. Note that this time includes the execution of grammar actions that create Postgres' parse tree. cpp-libpeg takes on average 266 ms to parse the test file. However, our experimental parser does not have grammar actions defined yet. When simulating actions by generating default AST actions for every rule, parsing time increases to 339 ms. Note that the AST generation is more expensive than required, because a node is created for each matching rule, even if there is no semantic meaning in the grammar at hand.

Overall, we can observe a ca. 10× slowdown in parsing performance when using the cpp-libpeg parser. However, it should be noted that the *absolute duration* of those two processes is still tiny; at least for analytical queries, sub-millisecond parsing time is more than acceptable as parsing still only accounts for a tiny fraction of overall query processing time. Furthermore, there are still ample optimization opportunities in the experimental parsers we created using an off-the-shelf PEG library. For example, the library makes

[3] https://github.com/yhirose/cpp-peglib
[4] https://github.com/hannes/peg-parser-experiments

heavy use of recursive function calls, which can be optimized e.g. by using a loop abstraction.

In the following, we present some experiments in extending the prototype parser with support for new statements, entirely new syntax and with improvements in error messages:

Adding the `UNPIVOT` Statement. In order to add `UNPIVOT`, we have to define a grammar rule and then modify `SingleStmt` to allow the statement in a global sequence of SQL statements. This is shown below. We define the new `UnpivotStatement` grammar rule by adding it to the dictionary, and we then modify the `SingleStmt` rule entry in the dictionary to also allow the new statement.

```
UnpivotStatement <- 'UNPIVOT' Identifier
    'ON' Parens(List(Identifier) / '*')

SingleStmt <- SelectStatement / UnpivotStatement
```

Note that we re-use other machinery from the grammar like the `Identifier` rule as well as the `Parens()` and `List()` macros to define the `ON` clause. The rest of the grammar dictionary remains unchanged. After modification, the parser can be re-initialized in another 3 ms. Parser execution time was unaffected.

Extending `SELECT` with `GRAPH_TABLE`. Below we show a small set of grammar rules that are sufficient to extend our experimental parser with support for the SQL/PGQ `GRAPH_TABLE` clause and the containing property graph patterns. With this addition, the parser can parse the query in Figure 5. Parser construction and parser execution timings were unaffected.

```
Name <- (Identifier? ':' Identifier) / Identifier
Edge <- ('-' / '<-') '[' Name ']' ('->' / '-')
Pattern <- Parens(Name WhereClause?) Edge
   Parens(Name WhereClause?)
PropertyGraphReference <- 'GRAPH_TABLE'i '('
        Identifier ','
        'MATCH'i List(Pattern)
        'COLUMNS'i Parens(List(ColumnReference))
    ')' Identifier?

TableReference <-
    PropertyGraphReference / ...
```

Adding `dplyr` Syntax Support. `dplyr`, the "Grammar of Data Manipulation", is the de facto standard data transformation language in the R Environment for Statistical Computing [18]. The language uses function calls and a special chaining operator (`%>%`) to combine operators. Below is an example dplyr query:

```
df %>%
  group_by(species) %>%
  summarise(
    n = n(),
    mass = mean(mass, na.rm = TRUE)
  ) %>%
  filter(n > 1, mass > 50)
```

With an extensible parser, it is feasible to add support for query languages like `dplyr` to a SQL parser.

Below is a simplified grammar snippet that enables our SQL parser to accept the `dplyr` example from above.

```
DplyrStatement <- Identifier Pipe Verb (Pipe Verb)*
Verb <- VerbName Parens(List(Argument))
VerbName <- 'group_by' / 'summarise' / 'filter'
Argument <- Expression / (Identifier '=' Expression)
Pipe <- '%>%'

SingleStmt <- SelectStatement /
    UnpivotStatement / DplyrStatement
```

It is important to note that the rest of the experimental SQL parser *still works*, i.e. the `dplyr` syntax now *also* works. Parser construction and parser execution timings were again unaffected.

Better Error Messages. As mentioned above, PEG parsers are able to generate better error messages elegantly [11]. A common novice SQL user mistake is to mix up the order of keywords in a query, for example, the `ORDER BY` must come after the `GROUP BY`. Assume an inexperienced user types the following query:

```
SELECT customer, SUM(sales)
FROM revenue
ORDER BY customer
GROUP BY customer;
```

By default, both the YACC and the PEG parsers will report a similar error message about an "unexpected 'GROUP' keyword" with a byte position. However, with a PEG parser we can define a "recovery" syntax rule that will create a useful error message. We modify the `OrderByClause` from Figure 6 like so:

```
OrderByClause <- 'ORDER'i 'BY'i List(OrderByExpression)
    %recover(WrongGroupBy)?
WrongGroupBy <- GroupByClause
    { error_message "GROUP BY must precede ORDER BY" }
```

Here, we use the `%recover` construct to match a misplaced `GROUP BY` clause, re-using the original definition, and then trigger a custom error message that advises the user on how to fix their query. And indeed, when we parse the wrong SQL example, the parser will output the custom message.

5 CONCLUSION AND FUTURE WORK

In this paper, we have proposed to modernize the ancient art of SQL parsing using more modern parser generators like PEG. We have shown how by using PEG, a parser can be extended at run-time at minimal cost without re-compilation. In our experiments we have demonstrated how minor grammar adjustments can fundamentally extend and change the accepted syntax.

An obvious next step is to address the observed performance drawback observed in our prototype. Using more efficient implementation techniques, it should be possible to narrow the gap in parsing performance between YACC-based LALR parsers and a

dynamic PEG parser. Another next step is to address some detail questions for implementation: For example, parser extension load order should ideally not influence the final grammar. Furthermore, while parser actions can in principle execute arbitrary code, they may have to be restrictions on return types and input handling.

We plan to switch DuckDB's parser, which started as a fork of the Postgres YACC parser, to a PEG parser in the near future. As an initial step, we have performed an experiment where we found that it is possible to interpret the current Postgres YACC grammar with PEG. This should greatly simplify the transitioning process, since it ensures that the same grammar will be accepted in both parsing frameworks.

ACKNOWLEDGMENTS

We would like to thank **Torsten Grust** and **Gábor Szárnyas** for their valuable suggestions. We would also like to thank **Carlo Piovesan** for his translation of the Postgres YACC grammar to PEG. Hannes Mühleisen is also affiliated with Centrum Wiskunde & Informatica (CWI) and Radboud Universiteit Nijmegen. Hannes Mühleisen gratefully acknowledges funding from the Dutch Research Council (NWO), project "Responsible Decentralized Data Architectures".

REFERENCES

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.
[2] R. J. Atwal, Peter A. Boncz, Ryan Boyd, et al. 2024. MotherDuck: DuckDB in the cloud and in the client. In *CIDR*.
[3] Bison authors. 2024. GNU Bison 3.8.1 Documentation Chapter 5.2: Shift/Reduce Conflicts. https://www.gnu.org/software/bison/manual/html_node/Shift_002fReduce.html.
[4] Franklin DeRemer. 1969. *Practical translators for LR(k) languages.* Ph. D. Dissertation. Massachusetts Institute of Technology, USA.
[5] Alin Deutsch, Nadime Francis, Alastair Green, et al. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *SIGMOD*. ACM, 2246–2258.
[6] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *POPL*. ACM, 111–122.
[7] Abigale Kim. 2024. *Survey and Evaluation of Database Management System Extensibility.* Technical Report CMU-CS-23-144. Carnegie Mellon University. https://www.pdl.cmu.edu/PDL-FTP/Database/CMU-CS-23-144.pdf.
[8] Donald E. Knuth. 1965. On the Translation of Languages from Left to Right. *Inf. Control.* 8, 6 (1965), 607–639.
[9] Nicolas Laurent and Kim Mens. 2015. Parsing expression grammars made practical. In *SLE*. ACM, 167–172.
[10] Tony Mason and Doug Brown. 1990. *lex & yacc.* O'Reilly & Associates, Inc., USA.
[11] Sérgio Medeiros and Fabio Mascarenhas. 2018. Syntax error recovery in parsing expression grammars. In *SAC* (Pau, France) *(SAC '18)*. Association for Computing Machinery, New York, NY, USA, 1195–1202.
[12] Sérgio Medeiros, Fabio Mascarenhas, and Roberto Ierusalimschy. 2014. Left recursion in Parsing Expression Grammars. *Sci. Comput. Program.* 96 (2014), 177–190.
[13] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL and a Proposal Towards a Simple and Expressive Query Language. In *CIDR*.
[14] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An Embeddable Analytical Database. In *SIGMOD*.
[15] Jeff Shute et al. 2024. SQL has problems. We can fix them: Pipe syntax in SQL. *Proc. VLDB Endow.* 17, 12 (2024), 4051–4063.
[16] Alexander van Renen, Dominik Horn, Pascal Pfeil, et al. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. In *VLDB 2024*.
[17] Guido van Rossum, Pablo Galindo, and Lysandros Nikolaou. 2020. PEP 617 – New PEG parser for CPython. https://peps.python.org/pep-0617/.
[18] Hadley Wickham, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *dplyr: A Grammar of Data Manipulation.* R package version 1.1.4, https://github.com/tidyverse/dplyr.