



# Advanced Query Manipulation with MySQL Proxy

Kay Röpke

MySQL Enterprise Tools  
Sun Microsystems

Innovation Everywhere

# State of the Proxy

- Under active development on Launchpad
- 2 devs, 1 QA (and growing the team)
- Basis for the Query Analyzer of MySQL Enterprise Monitor
- Think of it as “middleware”, not a “proxy”



# Common use-cases for Proxy

- Connection-based:
  - Load-balancing
  - Accounting
- Query-based:
  - Accounting
  - Sharding
  - Ensuring that queries are “safe”
  - Query Analysis
  - Read/Write splitting

# Common denominator

- Query-based use cases need
  - client + server metadata
  - implicit connection state (sql\_mode etc)
- But even having all that:
  - Hard to reliably fiddle with the query text.
  - Simplistic tokenizer brings you suprisingly far.
  - But still not far enough!



# Limitations (for 3 popular proxies)

- DPM (Dormando's Proxy for MySQL)
  - no tokenizer (I couldn't find it...)
  - hence no parser
- Spock Proxy
  - flex-based tokenizer from MySQL Proxy 0.6.x, some fixes not ported
  - “fuzzy parser”
- MySQL Proxy
  - flex-based tokenizer, has known shortcomings
  - minimal handwritten parser in Lua

# New approach

- Requirements, solution must
  - be portable, even across languages
  - be easy to debug
  - embeddable in other applications
- Since I'm biased, I picked ANTLR
  - Another Tool for Language Recognition

# Very brief ANTLR overview

- Can generate code for many languages
  - Java, C, C#, Python...
- has an IDE, with a visual debugger
- Integrates lexers, parsers and tree-parsers with one grammar language
- Allows for arbitrary actions (= custom code in the grammar)
- Recursive Descent parsing strategy, with infinite lookahead possible (called LL(\*))

# ANTLR & MySQL Proxy



- Realization was:
  - For a reliable query manipulation we *need* a full parser (and lexer!)
  - Adapting the ones from mysqld was not particularly attractive
    - Lexer is handwritten
    - Parser is in yacc<sup>H</sup>H<sup>H</sup>H<sup>H</sup> yacc.
  - The emphasis is on
    - maintainability
    - clarity
    - extensibility

# What do we want?

- Ability to
  - tokenize a query
  - parse the query
    - we need the syntax tree
    - we need the symbols (tables, columns etc)
  - work with the tree
    - get subtrees
    - replace subtrees
    - stringify (sub-)trees



# Tokenization

- Some Lua code (from the examples)

```
local recognizer = parser.new(query)
local tokens = recognizer.tokens

-- just for kicks, print all tokens
for i = 1, #tokens do
    -- dump the token's attributes
    local token = tokens[i]
    dump_token(token)
end
```

# What's in a token?

- name
- index (into the token stream)
- text
- channel
- line (of first character)
- position\_in\_line (again, first character)

# Channel?

- ANTLR is like a radio
- Tokens can be placed on a channel
- Parsers “tune” into channels
- Great for tokens that you
  - want to preserve
  - but that don’t carry semantic significance
- whitespace
- comments (but not all of them)

# Parsing the query

- more from the example Lua script

```
local stmt_tree = recognizer.tree  
  
print("statement tree in Lisp-like  
notation")  
print(stmt_tree.treeString)  
  
print("text of statement (entire  
tree)")  
print(stmt_tree.text)
```

# And what now?

- Parsing alone is not enough
  - All it does is to verify that the input forms a “sentence” in the grammar
  - We do not perform any actions during parsing!
  - Instead we build a tree (AST = Abstract Syntax Tree)



# Trees

- Luckily ANTLR directly supports trees
- Usually you hand-code visitors for the trees
  - Example: mysqld's Item tree
- We don't want to handcode!
- Enter ANTLR's treeparsers

# Treeparsers

- Grammar for trees (validates tree input)
- Parsers consume 1-dimensional stream of tokens
- Treeparsers consume 2-dimensional stream of tree nodes
  - Actually: It's still 1-dimensional, but includes "UP"/"DOWN" nodes

# How can we make use of them?

- Different approaches:
  - Specialized treeparsers for each different application
  - Generic treeparser “doing it all”
- Problem: We do not know what you want to do.



# A generic treeparser

- Wouldn't it be nice to have callbacks?

```
local callbacks = {
  show_engine =
    function(name, attr)
      print("`" .. attr.text ..
        "` for engine `" ..
        name.text .. "`")
    end
}
stmt_tree["callbacks"] = callbacks
print("walking tree")
stmt_tree.walk
```



# Demo/Example