Aquery to Q Compiler: Parser Grammar

Jose Cambronero

February 12, 2015

1 Introduction

As part of implementing a compiler from Aquery to q[1], we have developed the BNF grammar below. This grammar is eventually used in implementing a flex/bison parser for Aquery.

2 Grammar

An Aquery program consists on a top level composed by queries, table/view creation, insert/update/delete statements. An empty program is also a valid aquery program as noted by the epsilon production for top-level.

2.1 Top-level program

⟨program⟩ ::= ⟨top_level⟩

⟨top_level⟩ ::= ⟨global_query⟩ ⟨top_level⟩
| ⟨create_table_or_view⟩ ⟨top_level⟩
| ⟨insert_statement⟩ ⟨top_level⟩
| ⟨update_statement⟩ ⟨top_level⟩
| ⟨delete_statement⟩ ⟨top_level⟩
| ⟨user_function_definition⟩ ⟨top_level⟩
| ϵ (*note: an empty aquery program is still an aquery program*)

2.2 Local and global queries

We proceed to define what constitutes a global and local query, those that can solely be used within queries between the **WITH** keyword and the following global query. Note that aside from the necessary declarations at the beginning, the remainder of the query is a normal query, and thus refers to the grammar rule associated with the base_query non-terminal.

⟨global_query⟩ ::= ⟨local_queries⟩ ⟨base_query⟩

[1] Aquery is an ordered database query language developed by Alberto Lerner and Dennis Shasha, for more information please see https://cs.nyu.edu/web/Research/TechReports/TR2003-836/TR2003-836.pdf

⟨*local_queries*⟩ ::= ’WITH’ ⟨*local_query*⟩ ⟨*local_queries_tail*⟩
| ϵ

⟨*local_queries_tail*⟩ ::= ⟨*local_query*⟩ ⟨*local_queries_tail*⟩
| ϵ

⟨*local_query*⟩ ::= ⟨*identifier*⟩ ⟨*col_aliases*⟩ ’AS’ ’(’ ⟨*base_query*⟩ ’)’

⟨*col_aliases*⟩ :: = ’(’ ⟨*comma_identifier_list*⟩ ’)’
| ϵ

⟨*comma_identifier_list*⟩ ::= ⟨*identifier*⟩ ⟨*comma_identifier_list_tail*⟩

⟨*comma_identifier_list_tail*⟩ ::= ’,’ ⟨*identifier*⟩ ⟨*comma_identifier_list_tail*⟩
| ϵ

⟨*column_list*⟩ ::= ⟨*column_name*⟩ ⟨*column_list_tail*⟩

⟨*column_name*⟩ ::= ⟨*identifier*⟩ | ⟨*column_dot_access*⟩

⟨*column_dot_access*⟩ ::= ⟨*identifier*⟩ ’.’ ⟨*identifier*⟩

⟨*column_list_tail*⟩ ::= ’,’ ⟨*column_name*⟩ ⟨*column_list_tail*⟩
| ϵ

2.3 Base query

A query requires a select clause and a from clause. There are additional optional clauses including an ordering clause (the base of declarative order in Aquery), a where clause, a group by clause, and a having clause, which can be used solely in conjunction with a group by clause.

⟨*base_query*⟩ ::= ⟨*select_clause*⟩ ⟨*from_clause*⟩ ⟨*order_clause*⟩ ⟨*where_clause*⟩ ⟨*groupby_clause*⟩

⟨*select_clause*⟩ ::= ’SELECT’ ⟨*select_elem*⟩ ⟨*select_clause_tail*⟩

⟨*select_elem*⟩ ::= ⟨*value_expression*⟩ ’as’ ⟨*identifier*⟩
| ⟨*value_expression*⟩

⟨*select_clause_tail*⟩ ::= ’,’ ⟨*select_elem*⟩ ⟨*select_clause_tail*⟩
| ϵ

⟨*from_clause*⟩ ::= ’FROM’ ⟨*table_expressions*⟩

⟨*order_clause*⟩ ::= ’ASSUMING’ ⟨*order_specs*⟩
| ϵ

⟨*order_specs*⟩ ::= ⟨*order_spec*⟩ ⟨*order_specs_tail*⟩

⟨*order_spec*⟩ ::= ’ASC’ ⟨*column_name*⟩
| ’DESC’ ⟨*column_name*⟩

⟨*order_specs_tail*⟩ ::= ’,’ ⟨*order_spec*⟩ ⟨*order_specs_tail*⟩
| ϵ

⟨*where_clause*⟩ ::= ’WHERE’ ⟨*search_condition*⟩
| ϵ

⟨*groupby_clause*⟩ ::= ’GROUP’ ’BY’ ⟨*comma_value_expression_list*⟩ ⟨*having_clause*⟩
| ϵ

⟨*having_clause*⟩ ::= ’HAVING’ ⟨*search_condition*⟩
| ϵ

2.3.1 Search condition

`where` clauses, along with other clauses such as `having` and `on`, require a search condition non-terminal. We borrow the notion of a search condition from SQL 92 grammar, but also allow traditional value expressions to be a member of a search condition, which allows user defined functions to be used as stand-alone predicates, as well as boolean-typed columns. Please see http://savage.net.au/SQL/sql-92.bnf.html#searchcondition for the SQL92 details on this.

⟨*search_condition*⟩ ::= ⟨*boolean_term*⟩
| ⟨*search_condition*⟩ ’OR’ ⟨*boolean_term*⟩

⟨*boolean_term*⟩ ::= ⟨*boolean_factor*⟩
| ⟨*boolean_term*⟩ ’AND’ ⟨*boolean_factor*⟩

⟨*boolean_factor*⟩ ::= ⟨*boolean_primary*⟩
| ’NOT’ ⟨*boolean_primary*⟩

⟨*boolean_primary*⟩ ::= ⟨*predicate*⟩
| ’(’ ⟨*search_condition*⟩ ’)’

⟨*predicate*⟩ ::= ⟨*value_expression*⟩ ⟨*posfix_predicate*⟩
| ⟨*overlaps_predicate*⟩

⟨*posfix_predicate*⟩ ::= ⟨*between_predicate*⟩
| ⟨*in_predicate*⟩
| ⟨*like_predicate*⟩
| ⟨*null_predicate*⟩
| ⟨*is_predicate*⟩
| ϵ (* value_expression is already boolean *)

⟨*between_predicate*⟩ ::= BETWEEN ⟨*value_expression*⟩ ’AND’ ⟨*value_expression*⟩
| ’NOT’ ’BETWEEN’ ⟨*value_expression*⟩ ’AND’ ⟨*value_expression*⟩

⟨*in_predicate*⟩ ::= ’IN’ ⟨*in_pred_spec*⟩
| ’NOT’ ’IN’ ⟨*in_pred_spec*⟩

⟨*in_pred_spec*⟩ ::= ⟨*value_expression*⟩ (* implicit list *)
| ’(’ ⟨*comma_value_expression_list*⟩ ’)’

⟨*like_predicate*⟩ ::= ’LIKE’ ⟨*value_expression*⟩
| ’NOT’ ’LIKE’ ⟨*value_expression*⟩

⟨*null_predicate*⟩ ::= ’IS’ ’NULL’
| ’IS’ ’NOT’ ’NULL’

⟨*is_predicate*⟩ ::= ’IS’ ⟨*truth_value*⟩
| ’IS’ ’NOT’ ⟨*truth_value*⟩

⟨*truth_value*⟩ ::= ’TRUE’ | ’FALSE’

⟨*overlaps_predicate*⟩ ::= ⟨*range_value_expression*⟩ ’OVERLAPS’ ⟨*range_value_expression*⟩

⟨*range_value_expression*⟩ ::= ’(’ ⟨*value_expression*⟩ ’,’ ⟨*value_expression*⟩ ’)’

2.3.2 Table Expressions

We now proceed to define what table expressions constitute. Informally, table expression can be an identifier associated with a table, or an operation on a table (such as flatten), or a join on tables. Join grammar inspired by http://savage.net.au/SQL/sql-92.bnf.html#joinedtable. Note that the precedence/associativity of joins and other operations is directly encoded in the grammar.

⟨*table_expressions*⟩ ::= ⟨*joined_table*⟩ ⟨*table_expressions_tail*⟩

⟨*table_expressions_tail*⟩ ::= ’,’ ⟨*joined_table*⟩ ⟨*table_expressions_tail*⟩ (*note the semantics of this are cross join*)
| ϵ

⟨*joined_table*⟩ ::= ⟨*table_expression*⟩ ⟨*join_type*⟩ ’JOIN’ ⟨*joined_table*⟩ ⟨*join_spec*⟩

⟨*join_type*⟩ ::= ’INNER’ | ’FULL’ ’OUTER’

⟨*join_spec*⟩ ::= ⟨*on_clause*⟩ | ⟨*using_clause*⟩

⟨*on_clause*⟩ ::= ’ON’ ⟨*search_condition*⟩

⟨*using_clause*⟩ ::= ’USING’ ’(’ ⟨*comma_identifier_list*⟩ ’)’

⟨*table_expression*⟩ ::= ⟨*table_expression_main*⟩
| ⟨*built_in_table_fun*⟩ ’(’ ⟨*table_expression_main*⟩ ’)’

⟨*table_expression_main*⟩ ::= ⟨*identifier*⟩ ⟨*identifier*⟩ (* implicit correlation name *)
| ⟨*identifier*⟩ ’AS’ ⟨*identifier*⟩ (* explicit correlation name *)
| ⟨*identifier*⟩
| ’(’ ⟨*joined_table*⟩ ’)’

⟨*built_in_table_fun*⟩ ::= ’FLATTEN’ (* potentially more to add here *)

2.4 Table and View Creation

We define table and view creation at the top-level

⟨*create_table_or_view*⟩ ::= 'CREATE' 'TABLE' 'ID' ⟨*create_spec*⟩
| 'CREATE' 'VIEW' 'ID' ⟨*create_spec*⟩

⟨*create_spec*⟩ ::= 'AS' ⟨*global_query*⟩
| '(' ⟨*schema*⟩ ')'

⟨*schema*⟩ ::= ⟨*schema_element*⟩ ⟨*schema_tail*⟩

⟨*schema_element*⟩ ::= ⟨*identifier*⟩ ⟨*type*⟩

⟨*schema_tail*⟩ ::= ',' ⟨*schema_element*⟩ ⟨*schema_tail*⟩
| ϵ

⟨*type*⟩ ::= 'INT' |'FLOAT' |'STRING' | 'DATE' | 'BOOLEAN' | 'HEX'

2.5 Updating, Inserting, Deleting

We define the grammar relating to update, insert and delete statements at the top level.

⟨*update_statement*⟩ ::= 'UPDATE' ⟨*identifier*⟩ 'SET' ⟨*set_clauses*⟩ ⟨*order_clause*⟩ ⟨*where_clause*⟩ ⟨*groupby_clause*⟩

⟨*set_clauses*⟩ ::= ⟨*set_clause*⟩ ⟨*set_clauses_tail*⟩

⟨*set_clauses_tail*⟩ ::= ',' ⟨*set_clause*⟩ ⟨*set_clauses_tail*⟩
| ϵ

⟨*set_clause*⟩ ::= ⟨*identifier*⟩ '=' ⟨*value_expression*⟩

⟨*insert_statement*⟩ ::= 'INSERT' 'INTO' ⟨*identifier*⟩ ⟨*order_clause*⟩ ⟨*insert_modifier*⟩ ⟨*insert_source*⟩

⟨*insert_modifier*⟩ ::= '('⟨*comma_identifier_list*⟩ ')' (* insert values into given order of column names *)
| ϵ (* insert into default column order *)

⟨*insert_source*⟩ ::= ⟨*global_query*⟩
| 'VALUES' '(' ⟨*comma_value_expression_list*⟩ ')'

⟨*delete_statement*⟩ ::= 'DELETE' ⟨*from_clause*⟩ ⟨*order_clause*⟩ ⟨*where_clause*⟩
| 'DELETE' ⟨*comma_identifier_list*⟩ ⟨*from_clause*⟩ (* similarly to q, we can choose to delete rows where the predicates in where_clause are true, or we can choose to delete a column, but not both. No need to specify order, since will delete whole column...*)

2.6 User Defined Functions

We now define another element of the top-level: user defined function. Functions can have a series of expressions, queries, or local variable definitions. All but the last of which have to be followed by a semi-colon. The result of the function is the last expression evaluated.

⟨*user_function_definition*⟩ ::= ’FUNCTION’ ⟨*identifier*⟩ ’(’ ⟨*comma_identifier_list*⟩
’)’ ’{’ ⟨*function_body*⟩ ’}’

⟨*function_body*⟩ ::= ⟨*function_body_elem*⟩ ⟨*function_body_tail*⟩
| ϵ (*note: a function with no body is still a function *)

⟨*function_body_tail*⟩ ::= ’;’ ⟨*function_body_elem*⟩ ⟨*function_body_tail*⟩
| ϵ

⟨*function_body_elem*⟩ ::= ⟨*value_expression*⟩ | ⟨*function_local_var_def*⟩ | ⟨*local_queries*⟩
⟨*base_query*⟩ (* functions can have expressions or queries *)

⟨*function_local_var_def*⟩ ::= ⟨*identifier*⟩ ’:=’ ⟨*value_expression*⟩

2.7 Value Expressions

We encode operator precedence and associativity into the grammar itself. This section of the grammar draws inspiration from `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`. Some expressions also draw inspiration from `http://savage.net.au/SQL/sql-92.bnf.html`.

⟨*constant*⟩ ::= ⟨*integer*⟩ | ⟨*float*⟩ | ⟨*date*⟩ | ⟨*string*⟩ | ⟨*hex*⟩ | ⟨*truth_value*⟩

⟨*table_constant*⟩ ::= ’ROWID’ | ⟨*column_dot_access*⟩ | ’*’

⟨*case_expression*⟩ ::= ’CASE’ ⟨*case_clause*⟩ ⟨*when_clauses*⟩ ⟨*else_clause*⟩ ’END’
(* sql92 *)

⟨*case_clause*⟩ ::= ⟨*value_expression*⟩ (* value is compared to values in when clauses*)
| ϵ (* when clause is treated as a search *)

⟨*when_clauses*⟩ ::= ⟨*when_clause*⟩ ⟨*when_clauses_tail*⟩

⟨*when_clauses_tail*⟩ ::= ⟨*when_clause*⟩ ⟨*when_clauses_tail*⟩
| ϵ

⟨*when_clause*⟩ ::= ’WHEN’ ⟨*search_condition*⟩ ’THEN’ ⟨*value_expression*⟩ (*search condition generalizes to the value-comparison case expression, since a search condition non-terminal can expand into a simple value expression *)

⟨*else_clause*⟩ ::= ’ELSE’ ⟨*value_expression*⟩
| ϵ

```
⟨main_expression⟩ ::= ⟨constant⟩ | ⟨table_constant⟩ | ⟨identifier⟩ | '(' ⟨value_expression⟩
      ')' | ⟨case_expression⟩

⟨call⟩ ::= ⟨main_expression⟩
  |  ⟨main_expression⟩ '[' ⟨indexing⟩ ']'
  |  ⟨built_in_fun⟩ '(' ')'
  |  ⟨built_in_fun⟩ '(' ⟨comma_value_expression_list⟩ ')'
  |  ⟨identifier⟩ '(' ')' (* user defined functions *)
  |  ⟨identifier⟩ '(' ⟨comma_value_expression_list⟩ ')' (*user defined functions*)

⟨indexing⟩ ::= ODD | EVEN | EVERY ⟨integer⟩

⟨built_in_fun⟩ ::= 'abs' | 'avg' | 'count' | 'deltas' | 'distinct' | 'drop' | 'fill' |
      'first' | 'last' | 'max' | 'maxs' | 'min' | 'mins' | 'mod' | 'next' | 'not'| 'prev' |
      'prd' | 'prds' | 'reverse' | 'sum' | 'sums' | 'stddev'|

⟨exp_expression⟩ ::= ⟨call⟩
  |  ⟨call⟩ ⟨exp_op⟩ ⟨exp_expression⟩

⟨mult_expression⟩ ::= ⟨exp_expression⟩
  |  ⟨mult_expression⟩ ⟨times_op⟩ ⟨exp_expression⟩
  |  ⟨mult_expression⟩ ⟨div_op⟩ ⟨exp_expression⟩

⟨add_expression⟩ ::= ⟨mult_expression⟩
  |  ⟨add_expression⟩ ⟨plus_op⟩ ⟨mult_expression⟩
  |  ⟨add_expression⟩ ⟨minus_op⟩ ⟨mult_expression⟩

⟨rel_expression⟩ ::= ⟨add_expression⟩
  |  ⟨rel_expression⟩ ⟨l_op⟩ ⟨add_expression⟩
  |  ⟨rel_expression⟩ ⟨g_op⟩ ⟨add_expression⟩
  |  ⟨rel_expression⟩ ⟨le_op⟩ ⟨add_expression⟩
  |  ⟨rel_expression⟩ ⟨ge_op⟩ ⟨add_expression⟩

⟨eq_expression⟩ ::= ⟨rel_expression⟩
  |  ⟨eq_expression⟩ ⟨eq_op⟩ ⟨rel_expression⟩
  |  ⟨eq_expression⟩ ⟨neq_op⟩ ⟨rel_expression⟩

⟨and_expression⟩ ::= ⟨eq_expression⟩
  |  ⟨and_expression⟩ ⟨and_op⟩ ⟨eq_expression⟩

⟨or_expression⟩ ::= ⟨and_expression⟩
  |  ⟨or_expression⟩ ⟨or_op⟩ ⟨and_expression⟩

⟨value_expression⟩ ::= ⟨or_expression⟩
```

Now that we have value expressions defined, we define a form of value expression list: comma separated value expressions.

```
⟨comma_value_expression_list⟩ ::= ⟨value_expression⟩ ⟨comma_value_expression_list_tail⟩

⟨comma_value_expression_list_tail⟩ ::= ',' ⟨value_expression⟩ ⟨comma_value_expression_list_tail⟩
  |  ϵ
```

This concludes the formal outline of the Aquery grammar. Note that this grammar maybe revised and changed as necessary throughout development if need be.

For a flex/bison implementation of this grammar please see https://www.github.com/josepablocam/aquery2q/parser/