Aquery to Q Compiler: Symbol Table, AST and Visualization

Jose Cambronero

March 4, 2015

1 Symbol Table

1.1 Overview

The compiler manages a simple symbol table to keep track of user-defined functions (UDF), local query temporary names and local variable definitions (permitted within user-defined function definitions). Given that aquery is meant to interact with an outside (and hence unknown) kdb environment, we don't necessarily use our symbol table to the same degree other traditional compilers might.

1.2 Uses

We will perform very "light" type checking that involves the symbol table. Namely, when we observe a function call, we can verify that the element that is acting as a function should be either unknown(i.e. possibly coming from an external source), a built-in or an UDF. Any other types can be reported as an error.

The main purpose of the table, however, is to provide meta information on functions. Namely, we keep track of what functions are order-dependent and/or size-preserving by creating entries with details in the symbol table.

1.3 Implementation

Our symbol table is managed as a stack of hash tables. Upon entering a new scope (e.g. the body of a function definition or a set of local queries), we push a new table onto our stack. Any identifiers that need to be tracked in that scope are hashed into the table. Upon exiting the scope, we simply pop the top table.

Each table currently consists of 32 positions. Hashcodes are calculated simply as the sum of the integer value of each character in an identifier and the result is modded with the size of the table. Any collisions are handled by chaining the records.

2 AST

2.1 Overview

The aquery parser builds an AST as it parses an aquery source file. Note that while the parser currently explicitly builds nodes for all statements in aquery, this will be changed so that only optimization sections undergo explicit AST construction (all others will be parts of implicit parse trees built by bison during the parsing process). We note that this strategy is preferable as the only goal of building an AST in our case is to manipulate the portions of the tree related to queries and their parts.

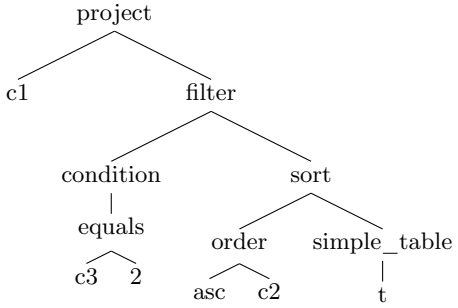
2.2 Philosophy

We attempt to mix traditional ASTs with a form of query plan. All non-query statements conform to traditional ASTs with operators represented as nodes and arguments as children of those nodes. Query related nodes, however, should be viewed as part of a logical query plan.

For example, the query

```
SELECT c1 FROM t ASSUMING ASC c2 WHERE c3 = 2
```

would become something akin to



The left child of each node represents any parameters needed, while the right side child represents the state of computation up to that point, so these nodes can be easily composed to reflect a plan. Accordingly, deeper nodes represent earlier steps in the plan.

2.3 Implementation

While we won't describe each and every type of AST node (there are many), there are 2 nodes worth describing given their importance for the optimization stages. Namely, the expression node and the logical query node. For code details please see ast.c and ast.h.

2.3.1 Expression Nodes

Expression nodes represent all the value expressions in the BNF aquery grammar, and additionally some other portions, such as search conditions, which are easily expressed as expression nodes. Expression nodes are built to create n-ary trees by providing a pointer to the first child and a pointer to the next sibling. Additionally, expression nodes have a union for explicit data, as identifiers and constants are stored in expression nodes for use in code generation. In order to compose expressions, we required

a lot of different types of value expressions to have the same struct type. Given this, we also include a field node_type that carries an enumeration of possible expression nodes and allows us to distinguish among all possibilities. Expression nodes also carry a data type field, that could be useful for some type checking later on. Additional information includes any order dependencies for that node, order dependencies for subtrees rooted at that node, and information on whether the node is sortable (i.e. can we index into it).

```
typedef struct ExprNode {
    ExprNodeType node_type;
    DataType data_type;
    int order_dep; //order dependent
    int sub_order_dep; //subtree has order deps
    int can_sort; //(e.g. we can sort c1 but not 10)
        union {
        int ival;
        float fval;
        char *str;
        }data;
    struct ExprNode *first_child;
    struct ExprNode *next_sibling;
} ExprNode;
```

2.3.2 Logical Query Nodes

As mentioned before, AST branches associated with queries should be viewed as part of a query plan. Thus nodes in these branches are called logical query nodes (as they relate to a logical query plan). Similarly to expression nodes, we need to compose many types of clauses (from/select/where etc) and be able to move them around freely for optimization. Given this, all parts of a query are labeled as logical query nodes but we provide a field, node_type, to distinguish among the possibilities. Each node has a union for potential parameters (e.g. a sorting node needs sorting specification, projection node needs expressions). Composition of logical query nodes is achieved by linking via an argument pointer (arg), which points to the previous step in the query plan (ie. the next deepest node in the branch). Some logical query operations, like join, require more than 1 argument, namely 2 tables. Given this, logical query nodes also have a pointer to another potential argument (next_arg), which can be followed in situations like this.

```
typedef struct LogicalQueryNode {
    LogicalQueryNodeType node_type;
    struct LogicalQueryNode *after; //we will move a lot of these around
    struct LogicalQueryNode *arg; //argument to operation composition
    struct LogicalQueryNode *next_arg; //potential additional arguments
    int order_dep;
    union {
        char *name                              ;
        NamedExprNode    *namedexprs;
        ExprNode                        *exprs;
        IDListNode                      *cols;
        OrderNode                       *order;
        } params;
} LogicalQueryNode;
```

3 Visualization

3.1 Overview

Visualizing an AST can be critical for a variety of reasons

1. Verify grammar
2. Debug
3. Visualize optimization and other transformations

The last point should prove to be the most valuable as we begin our optimization phase.

In order to visualize our AST, we leverage graphviz's dot program to perform the actual graphing of the tree. The aquery parser simply traverses the AST built during parsing and prints out a dot format of it to stdout. We chose dot for a variety of reasons, but the most salient are: its syntax is very simple and easily reproducible during a tree traversal, and it is very popular and thus there are additional utilities associated with dot files. Below a simple example of a query as displayed by the current printing capacity:

```
SELECT c1 FROM t WHERE c2 = 2
```