SQL Parse Event Walker

February 7, 2017 – Geoffrey A. Howe

# Introduction

The SQL Parse Event Walker collects and produces four different objects when parsing SQL statements. These include:

1. A nested Map<> object holding an Abstract Syntax Tree of the Query

2. A nested Symbol Table

3. A Source Table Dictionary containing the column names of the input tables

4. An output Interface list, containing the names of the output columns

# Abstract Syntax Tree of the Query

The raw parse tree is quite specific to the grammar and produces a data structure that is difficult to work with generally. In order to simplify working with the query text, the SqlParseEventWalker creates an Abstract Syntax Tree representation of the query built using nested HashMap<String, Object> objects. The syntax tree preserves the essential structure of the query, using a set of standard keys to represent language constructs and phrase types while holding the details as subtrees.

The AST can be walked recursively to explore different levels, or to apply transformations. A set of language generators will be developed for various SQL syntaxes (e.g., Hive, Postgresql, SPARQL, others as needed). These will support substitutions and aliasing when given a symbol table substitution object.

The AST Keys that will appear are documented in the following list. Some keys will have variable names. In some entries, examples of the child subtrees may be presented. Examples of the more complex situations (such as UNION and INTERSECTION) will also be presented.

## Predicand Subtrees

A predicand is any of a variety of statements defining or representing something that provides a singular value. For example, in the formula “A + B = C”, A, B and C are all predicands of the formula. In SQL, there are several kinds of predicands, some more complex than others. Each one is represented by a single map structure. The following table names and then shows the purpose and meaning of each type of predicand.

| Predicand Type | SQL Statement | Example |
| --- | --- | --- |
| Column Reference | course\_tab.subj\_code as EXTERNAL\_ID | {column={name=subj\_code, table\_ref=course\_tab}, alias=EXTERNAL\_ID } |
| Literal Value | ‘AA’ as EXTERNAL\_ID | {literal='AA', alias=EXTERNAL\_ID} |
| Null Value Literal | Null | {null\_literal=null} |
| Concatenation Value | a || b | { concatenate={1={column={name=a, table\_ref=null}}, 2={column={name=b, table\_ref=null}}}}} |
| Function Value | concat\_ws(‘-‘, crs.subject\_code, crs.course\_number) as EXTERNAL\_ID | {function={parameters={1={literal='-'}, 2={column={name=subject\_code, table\_ref=crs}}, 3={column={name=course\_number, table\_ref=crs}}}, function\_name=concat\_ws}, alias=EXTERNAL\_ID} |
| Aggregate Function Value | max(scbcrse\_eff\_term)  count(distinct scb.scbcrse\_id) | { function={function\_name=MAX, qualifier=null, parameters={column={name=scbcrse\_eff\_term, table\_ref=null}}}}  { function={function\_name=COUNT, qualifier=distinct, parameters={column={name=scbcrse\_id, table\_ref=scb}}}} |
| Case Function Value | case when true then ‘Y’ when false then ‘N’ else ‘N’ end as case\_one | {case={clauses={1={then={literal='Y'}, when={literal=true}}, 2={then={literal='N'}, when={literal=false}}}, else={literal='N'}}, alias=case\_one} |
| Window Function | rank() OVER (partition by k\_stfd, kppi order by OBSERVATION\_TM desc, row\_num desc) AS key\_rank | {alias=key\_rank, window\_function={over={partition\_by={1={column={name=k\_stfd, table\_ref=null}}, 2={column={name=kppi, table\_ref=null}}}, orderby={1={sort\_order=desc, column={column={name=OBSERVATION\_TM, table\_ref=null}}, null\_order=null}, 2={sort\_order=desc, column={column={name=row\_num, table\_ref=null}}, null\_order=null}}}, function={function\_name=rank, parameters=null}}} |
| Lookup Subquery | A subquery embedded as a predicand. | {lookup={select={1={column={name=item, table\_ref=null}}}, from={table={alias=null, table=sgbstdn}}, where={left={column={name=sgbstdn\_levl\_code, table\_ref=null}}, right={literal='US'}, operator==}}, alias=INTERNATIONAL\_IND}} |
| Arithmetic Expression | -(aa.scbcrse\_coll\_code \* 6 - other) as alias | {calc={left={literal=-1}, right={parentheses={calc={left={calc={left={column={name=scbcrse\_coll\_code, table\_ref=aa}}, right={literal=6}, operand=\*}}, right={column={name=other, table\_ref=null}}, operand=-}}}, operand=\*}, alias=alias} |

## Condition Subtrees

There are several locations where SQL allows condition statements (for example, where clauses, join clauses, case when statements). The AST produced will not explicitly signal that the subtree is a condition, but that knowledge should be implied by the context.

There are several variations of conditions that will appear in a condition subtree. Condition subtrees are nested, so that at each place in a condition subtree, another whole condition subtree might be substituted. In what follows, any ellipsis (e.g., “{…}”) represents a place where another condition subtree of any type could be inserted.

| Type of Condition | Example |
| --- | --- |
| Simple Comparison Condition | {condition={left={column={name=b, table\_ref=null}}, right={column={name=d, table\_ref=null}}, operator==}}} |
| Simple Boolean | {literal=true} |
| List of And Conditions | {and={1={…}, 2={…}}} |
| List of Or Conditions | {or={1={…}, 2={…}}} |
| Parenthetical Condition | {parentheses={…}} |
| Not Condition | {not={…}} |
| Between Condition | <? NOT WORKING CORRECTLY ?> |
| In Condition | {in={item={column={name=subj\_code, table\_ref=null}}, in\_list={list={1={literal='AA'}, 2={literal='BB'}}}}} |
| Exists Predicate Condition | <? NOT WORKING CORRECTLY ?> |
| Pattern Match Condition | <? NEEDS DOCUMENTATION ?> |
| Singleton Conditions (Is Null) | {condition={left={column={name=section\_name, table\_ref=s}} , operator=is null}}  {condition={left={column={name=scbcrse\_subj\_code, table\_ref=aa}}, operator=is not null}} |
| Boolean Predicand | Any predicand that resolves to a Boolean value can also appear in a condition clause subtree |

## AST KEYS

### alias

A subtree representing a leaf node in the AST. The "alias" key points to a string containing the alias name to be used in referencing the thing aliased in other parts of the query, or in the final result (depending on where the alias is defined in the overall query tree).

### and

This key points to a condition subtree represent a list of conditions to be “and-ed” together. All of the entries form siblings in a list of conditions, which themselves could contain nested, recursive other condition subtrees of any kind.

EXAMPLE:

and={1={…}, 2={…}, 3={…}}

### calc

This key points to a nested subtree representing an arithmetic calculation. It consists of three keys, representing respectively the left side of a calculation, the right side, and the operator. The left and right keys can hold another arithmetic expression, or any of the predicands. Operators are the normal arithmetic operators for addition, subtraction, multiplication, division, modulus.

EXAMPLE: calc={left={column={…}}, right={column={…}}, operator=+}}

### case

This key holds a subtree representing a SQL case statement. There are two styles of case statement.

The first variant contains two immediate children subtrees, one being the clauses subtree being a numbered list of the when-then statements of the case statement, and the other being the else clause, if any, of the case statement.

Each of the clause entries is a subtree referenced by an ordinal key that maintains the sequence of the statement in the SQL. Within these subtrees, there will be two subtrees, the first being a condition subtree under the when key (holding a condition subtree) and the second the subtree holding any of the predicands (e.g.,, literal value, column, function result) under the then key.

EXAMPLE:

case={clauses={1={then={literal='Y'}, when={literal=true}}, 2={then={literal='N'}, when={literal=false}}}, else={literal='N'}}},

The second variant contains the same two clauses and else subtrees, but also an additional item subtree key which will hold a predicand to be used in each of the when clauses. Whereas the first variant holds a complete condition in each when key subtree, the second variant sets up an implied condition between the predicand in the item subtree and another predicand in the when subtree (instead of a condition). This variant can only be used when the expectation is that the condition will be a simple equivalence check on the values of the two predicands.

EXAMPLE:

case={ item={column={name=col, table\_ref=null}}, clauses={1={then={literal='Y'}, when={literal='a'}}, 2={then={literal='N'}, when={literal='b'}}}, else={literal='N'}}},

### column

This is a key that holds a subtree containing a column reference. Column references come in two flavors, the first being an actual single column reference, the other being a reference to “all” columns of a table. The column subtree itself is one of the predicands (see above explanation) of a SQL query and can appear in many different locations in a query statement.

**COLUMN REFERENCE:** A subtree consists of an embedded map containing a single "name" and a single "table\_ref" entry. The "name" being the column name from one of the query's tables or sub-queries, and the "table\_ref" contains either an alias or the actual table name where the column name is defined.

EXAMPLE: column={name=scbcrse\_coll\_code, table\_ref=aa}

**INCLUDE-ALL COLUMN REFERENCE:** A subtree is a "column reference" where the contents of the name have been set to the universal column selector "\*".

EXAMPLE: column={name=\*, table\_ref=aa}

### concatenate

This is a special predicand representing a function which concatenates other predicands. The predicand formulation depends on the SQL variant, but the most common structure is an infix operator. The predicand subtree, however, presents as a numbered list of predicand subtrees.

EXAMPLES:

concatenate={1={column={name=a, table\_ref=null}}, 2={column={name=b, table\_ref=null}}}}

### condition

This key points to a nested subtree representing a boolean comparison. It consists of two or three keys, representing respectively the left and right sides of a comparison, and the comparison operator. Alternatively, there are a handful of comparisons to fixed values, such as “is null” and “is not null”, in which there will be only a left predicand subtree and the operator. The left and right keys can hold another condition expression, or any of the predicands. Operators are the normal boolean operators for equals, less or greater than, etc., as well as notable singleton comparisons to null.

EXAMPLE: condition={left={column={…}}, right={column={…}}, operator=<=}}

condition={left={column={…}} , operator=is null}}

### from

This key points to either a possibly nested, recursive, join list subtree, a single table subtree, or a list of table subtrees captured as an ordinal list.

EXAMPLES:

from={join={…}}

from={table={…}} <? IS THIS CORRECT ?>

from={1={table={…}}, 2={table={…}}}

### function

The function key is the head of a subtree containing the specification of a function statement. Its subtree has two immediate children, a function\_name key and a parameters key. The function\_name is the program code name of the function, and the parameters is a numbered list of predicands whose values must be passed to the function logic for execution. Functions always return a single, scalar value which is why a function can act as a predicand in its own right.

A second variant supporting aggregation functions is formed slightly differently. While it has a function\_name, it also has an optional “qualifier” key, often pointing to a null value, but which may hold a “distinct” qualifier on the aggregate function. For this variant, the parameters key of the function will not have a numbered list of predicands, but will have a single predicand subtree directly within it.

Example:

function={parameters={1={…}, 2={…}, 3={…}}, function\_name=datestr}

function={function\_name=COUNT, qualifier=null, parameters=\*}

function={function\_name=MAX, qualifier=null, parameters={column={name=scbcrse\_eff\_term, table\_ref=null}}}

function={function\_name=trim, parameters={qualifier=leading, trim\_character={literal='0'}, value={column={name=field1, table\_ref=null}}}}}

function={parameters={1={concatenate={1={literal='0'}, 2={column={name=field2, table\_ref=null}}}}, 2={literal='0'}}, function\_name=trim}}}

### function\_name

The function\_name key is a leaf node subtree containing the text name of a function.

Example:

function\_name=datestr

### in

This key contains a subtree consisting of the component parts of an in statement. It always contains two keys, the item subtree is the value, column, function, etc. which will be searched for and the in\_list subtree which will hold either a list of values, or a subquery that returns a list of values to be searched.

EXAMPLES:

in={item={column={name=subj\_code, table\_ref=null}}, in\_list={list={1={literal='AA'}, 2={literal='BB'}}}}

in={item={column={name=subj\_code, table\_ref=null}}, in\_list={query={…}}}}

### in\_list

This key contains a subtree that holds either a list of values, or a subquery that returns a list of values to be searched.

EXAMPLES:

in\_list={list={1={literal='AA'}, 2={literal='BB'}}}

## intersect

This key can point at two different things. First, it can point to a numbered list containing combinations of subquery subtrees and instances of the second type of interect subtree. Second, this will point to a subtree which represents the actual interect statement. The interect list subtree will have a query subtree for every query participating in the interection, and will also include a child interect subtree for every interect clause.

The interect list subtree is actually a form of subquery tree, and can appear anywhere that a query subtree can appear. It appears as an ordinal list, with keys capturing the statement order in the query. As per typical interect statement ordering, this will typically appear as an alternating list of a query subtree then interect statement subtree, then query subtree.

The interect statement subtree consists of at least one and possibly two keys. Each will have a leaf node operator key containing a token representing the type of interect. Optionally, this subtree will include a qualifier key which is either null or indicates the interect “all” option.

EXAMPLES:

interect ={1={select={…}}, 2={ interect ={qualifier=null, operator= interect }}, 3={select={…}}

interect ={qualifier=null, operator= interect }

### item

This key appears as a value inside of a few different subtrees, including an in subtree and one variation of a case subtree. It points to a predicand subtree whose value would be used in further processing by whichever statement it is part of.

EXAMPLE: item={column={name=subj\_code, table\_ref=null}}

### join

This key can point at two different things. First, it can point to a numbered list containing combinations of table subtrees and instances of the second type of join subtree. Second, this will point to a subtree which represents the actual join statement. The join list subtree will have a table subtree for every table in a from clause, and will also include a child join subtree for every join clause in the from clause. The join list subtree will only appear in the from subtree, if a join statement is found in that subtree. The join statement subtree will only appear in a join list subtree.

The join list subtree appears as an ordinal list, with keys capturing the statement order in the query. As per typical join statement ordering, this will typically appear as an alternating list of table subtree then join statement subtree, then table subtree.

The join statement subtree consists of at least one and possibly two keys. Every join statement will have a leaf node join key containing a token representing the type of join. Optionally, this subtree will include an “on” key containing a condition subtree. This condition subtree represents the conditional logic required to match records through the join operation, but otherwise looks like any condition subtree.

Join type tokens that are recognized include the following.

| Tokens for “JOIN” Type Keys | Statement | Usage |
| --- | --- | --- |
| join | join | Default, basic join statement with no qualifiers or variations. |
| left | left join | Indicates a left join should be applied between the preceding and following subtrees in the join list. |
| leftouter | left outer join | Indicates a left outer join should be applied between the preceding and following subtrees in the join list. |
| right | right join | Indicates a right join should be applied between the preceding and following subtrees in the join list. |
| rightouter | right outer join | Indicates a right outer join should be applied between the preceding and following subtrees in the join list. |
| full | full join | Indicates a full join should be applied between the preceding and following subtrees in the join list. |
| fullouter | full outer join | Indicates a full outer join should be applied between the preceding and following subtrees in the join list. |
| inner | inner join | Indicates a inner join should be applied between the preceding and following subtrees in the join list. |
| naturaljoin | natural join | Special type of join supported by some SQL dialects. May introduce an extra, optional AST key entry in the join statement subtree, called “join\_type”, which would hold the qualifier for the natural join statement. Examples “inner”, “left”, “right”, etc. |
| crossjoin | cross join | Indicates a cross join should be applied between the preceding and following subtrees in the join list. |
| unionjoin | union join | Indicates a union join should be applied between the preceding and following subtrees in the join list. |

EXAMPLES:

join={1={table={…}}, 2={table={…}}}

join={1={table={…}}, 2={join=join, on={left={column={name=a, table\_ref=null}}, right={column={name=b, table\_ref=null}}, operator==}}, 3={table={…}}}

join={1={table={…}}, 2={join=join, on={…}}, 3={table={…}}}

### left

Key points to a subtree containing one of the predicands. It is typically part of a condition subtree or an arithmetic expression subtree, and would appear holding the predicand subtree appearing first in a comparison condition, or as the only predicand in a simple, single condition like an is null comparison. It could also appear in a between condition pointing at the first value of the comparison.

EXAMPLE: left={column={name=subj\_code, table\_ref=null}}

### name

Key points to a leaf node key containing the name of something. Example names include especially column names.

EXAMPLE: name=scbcrse\_coll\_code

### not

This key points to a condition subtree, of any style or kind and can appear any place where a condition subtree can appear. In other words, the “not” subtree is a type of condition subtree.

EXAMPLE: not={left={column={…}}, right={column={…}}, operator==}}

### null\_literal

A type of predicand subkey that represents the SQL constant for “null” value. This will be the sole entry in the subtree map and will have no value (value will be null).

Example: null\_literal=null

### null\_order

The key is used as a leaf node reference inside of an order by subtree. It holds the token representing the how null values should be treated. In most cases this will not be filled. If set, it will either have the token FIRST or LAST, indicating where the null values should be placed in the sort result.

EXAMPLE: null\_order=FIRST

### on

This key points to a condition subtree used in the join clause. Any condition subtree may be inserted into the on key’s subtree.

EXAMPLE: on={left={column={…}}, right={column={…}}, operator==}}

### operator

This key points to a leaf node containing one of many different types of operators. Operators come in families, including the arithmetic operators used for formula calculations, the boolean operators used for comparisons, and a number of unary comparisons, such as “is null” where the comparison is to some standard singleton. These can appear in any calc subtree, or in any condition subtree.

EXAMPLES:

operator=+

operator=<

operator=is not null

### or

This key points to a condition subtree represent a list of conditions to be “or-ed” together. All of the entries form siblings in a list of conditions, which themselves could contain nested, recursive other condition subtrees of any kind.

EXAMPLE:

or={1={…}, 2={…}, 3={…}}

## order\_by

This key holds a typical order by subtree. It can appear as part of the outermost query tree, or can be embedded within the over subtree of a window function. It holds a numbered list of order by subclauses, representing predicands that the output should use to sort the results of a partition.

Each of the list entries will consist of three keys, the first a column key pointing to a predicand (this could be a column, or any of the other predicand types). The second holds a leaf node indicating the direction of the sort (e.g., ASC for ascending and DESC for descending). The third holds an optional designation for how to handle null values on the order by column (e.g., null\_order).

EXAMPLES:

orderby={1={sort\_order=desc, column={column={…}}, null\_order=null}, 2={sort\_order=desc, column={column={…}}, null\_order=null}}}

## over

This key is part of a window\_function subtree. It will hold three subtrees, including the partition\_by statement, the order\_by statement, and the window function.

EXAMPLES:

over={partition\_by={…}, orderby={…}, function={function\_name=rank, parameters=null}

### parameters

The key is the head of a subtree containing a numbered list of predicands whose values must be passed to the function logic for execution. The parameters would be passed in the order or place number of its key. This is typically only found in a function subtree as one of its two standard elements.

Example:

parameters={1={…}, 2={…}, 3={…}}

### parentheses

This key points to a condition subtree which would have been enclosed in a pair of parentheses in the original query statement.

EXAMPLE:

parentheses={or={1={…}, 2={…}, 3={…}}}

## partition\_by

This key is part of a window\_function subtree and basically holds the partition definition statement. The partition by is a list of columns (predicands) in an order that will drive the partitioning of the underlying data.

EXAMPLES:

partition\_by={1={…}, 2={…}}

### qualifier

This key points to a leaf node containing one of many different types of qualifiers. These can appear in a number of different contexts, such as unions, joins, intersections, aggregate functions, and possibly others.

EXAMPLES:

qualifier=distinct

qualifier=all

### right

Key points to a subtree containing one of the predicands. It is typically part of a condition subtree, and would appear holding the predicand subtree appearing second (or last) in a comparison condition. It could also appear in a between condition pointing at the second value of the comparison.

EXAMPLE: right={column={name=subj\_code, table\_ref=null}}

### SQL

Several variations exist for the SQL subtree. In the first the SQL key holds the topmost query object, representing the entire nested, recursive SQL AST. The second variation would hold two subtrees, the first being a with subtree containing a list of named subqueries, and the second being the query subtree representing the main query of the statemet which will refer to the named subqueries in the with statement. A third variation represents a union of subqueries, while a fourth represents an intersection. Insert and update variations also exist.

EXAMPLE:

SQL={query={select={…}, from={…}}}

SQL={with={upsert={…}}, query={select={…}, from={table={alias=null, table=upsert}}}}

***SQL={union={1={select={…}}, 2={union={…}}, 3={select={…}}}}***

***SQL={intersect={1={select={…}}, 2={intersect={…}}, 3={select={…}}}}***

### schema

The key is used as a leaf node reference inside of a table subtree. If present, it holds the name of the schema where the table is defined.

EXAMPLE: schema=h

### select

The key contains a list of output columns. The list is presented as a Map whose keys are numerals representing the order of the output columns. Each subtree of these index keys is an embedded map consisting of a "column" subtree and an optional "alias" subtree.

EXAMPLE: select={1={column={...}, alias=bb}, 2={column={...}}, 3={column={...}}}

### sort\_order

The key is used as a leaf node reference inside of an order by subtree. It holds the token representing the order/direction of sorting, either ascending or descending (e.g., ASC, DESC respectively).

EXAMPLE: sort\_order = ASC

### table

The key is either a leaf node containing the name of an actual table, or it is a subtree holding a table object consisting of another table key (a leaf node in this case) and an optional table alias. The table may have been qualified by a schema name, and if so that also appears in the subtree as another leaf node. Most typically the table subtree appears in from, join, insert and update subtrees.

EXAMPLES:

table=studentcoursework

table={alias=aa, table=studentcoursework}

table={schema=h, alias=null, table=5463\_77}

### table\_ref

This key is used to hold a leaf value, namely a table or table alias value, depending on where it appears in the query. This is a *reference* to a table or subquery used in a column reference within the query. The reference might be the actual table name, or it could be the alias defined for the table, but it is used to differentiate which structure or context a column originated from in the query.

EXAMPLE: A query referring to all of the columns in a table, such as “aa.\*”, would create an “include-all column reference” that looks as follows:

column={name=\*, table\_ref=aa}

## union

This key can point at two different things. First, it can point to a numbered list containing combinations of subquery subtrees and instances of the second type of union subtree. Second, this will point to a subtree which represents the actual union statement. The union list subtree will have a query subtree for every query participating in the union, and will also include a child union subtree for every union clause.

The union list subtree is actually a form of subquery tree, and can appear anywhere that a query subtree can appear. It appears as an ordinal list, with keys capturing the statement order in the query. As per typical union statement ordering, this will typically appear as an alternating list of a query subtree then union statement subtree, then query subtree.

The union statement subtree consists of at least one and possibly two keys. Each will have a leaf node operator key containing a token representing the type of union. Optionally, this subtree will include a qualifier key which is either null or indicates the union “all” option.

EXAMPLES:

union={1={select={…}}, 2={union={qualifier=null, operator=union}}, 3={select={…}}

union={qualifier=null, operator=union}

## where

This key points to a condition subtree consisting of and and or subtrees, or singular condition subtrees.

EXAMPLES:

where={and={…}}

where={or={…}}

where={ left={column{…}}, right={column={…}}, operator==} }

union={qualifier=null, operator=union}

## window\_function

This key points to a SQL windowing function statement. These statements are quite complicated subtrees, with a lot of detail. The window function produces one output value which is calculated over the partition generated over the underlying data. Several different windowing functions are supported including rank, … <? The rest?>

EXAMPLES:

window\_function={over={partition\_by={1={column={name=k\_stfd, table\_ref=null}}, 2={column={name=kppi, table\_ref=null}}}, orderby={1={sort\_order=desc, column={column={name=OBSERVATION\_TM, table\_ref=null}}, null\_order=null}, 2={sort\_order=desc, column={column={name=row\_num, table\_ref=null}}, null\_order=null}}}, function={function\_name=rank, parameters=null}}

## with

The with statement in SQL is typically used to provide a way to define subquery logic in such a way that it can be inserted and used multiple times without having to copy it. Hence, it represents a set of named, subqueries that are then accessed in possibly multiple complex ways by a master query.

The with key points to a list subtree containing a set of named queries. This is a slightly different construction from other parts of the AST, in that each of the keys within the top level map is actually not predefined by the AST (i.e., they are not standard tokens). Instead, each key will be the query alias or pseudo-table name referenced in the main query. When processing the with list, each key should be used as the table reference or alias name referred to in the main SQL query (which will be a sibling subtree within the SQL AST tree).

EXAMPLES:

with={table\_1={select={…}, from={…}}, table\_2={select={…}, from={…}}}

## Special AST Variables

The SQL Parse Event Walker introduces a set of special variable references representing places where substitutions and expansion can be inserted into a SQL statement. Each type of variable will have a limited scope of utility, but can appear in any part of the AST. Alternatively, certain subtrees of the AST can be extracted from a very specific query and be replaced with an appropriate instance of one of these AST variables.

There are five types of AST Variables defined, each substituting for a different portion of a tree. These are as follows:

Value Variables: used to hold one or more literal values, of any type. These can appear anywhere a literal value can appear, such as constant values in a select statement, or as comparison values in any of the conditions. These are recognized by having two leading “#” signs, a variable name, and a trailing “#” sign. For example: “##undergraduate student#”.

Entity Type Variables: used to hold a logical name given to a table. These can appear anywhere a table name might appear in the SQL. These are recognized by appearing contextually where a table name should appear, and by having square brackets enclosing them. For example: “[student demographics]”.

Attribute Variables: used to hold a logical name given to a column of a table. The relationship between a column and its table strictly defines the relationship of an Attribute Variable to its Entity Type Variable. These are recognized by appearing within the context of an Entity Type variable but wherever a column name or other predicand would appear. These also appear with square brackets. For example: [student demographics].[gender] shows the “gender” attribute of the “student demographics” entity type.

Condition Variables: used to hold a condition statement of arbitrary complexity. These are best used when the condition can be self-contained, but can include Entity Type and Attribute Variables as components. These are recognized by appearing wherever a condition statement might appear, and are surrounded by angle brackets. For example: <transfer student> or <active>

Population Variables: used to hold an entire subquery. The subquery can also be defined using the other variable types. These represent complex logic and can appear anywhere a filtering subquery might appear. They are recognized by being surrounded by curly brackets. For example: {undergraduate students}.

More details to follow.