SQL Parse Event Walker

December 12, 2018 – Geoffrey A. Howe

# Introduction

The SQL Parse Event Walker collects and produces five 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
5. A list of substitution variables contained within the query (optional)

|  | *Description* | *Grammar* | *AST Tree* | *Symbol Table* | *Query Interface* | *Table Dictionary* | *Substitution Variables* |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | Generate Interface list and proper Table Dictionary from UPDATE queries | X | X | NA | TBD | TBD | NA |
| 12 | Assign unknown symbols from UPDATE query source tables to correct source table in Symbol Tree | X | X | TBD | NA | TBD | NA |
| 13 | Generate Interface list from INSERT queries | X | X | NA | TBD | X | NA |
| 14 | Assign unknown symbols from INSERT query source tables to correct source table in Symbol Tree | X | X | TBD | NA | TBD | NA |
| 15\* | Fix parser to recognize scientific notation | TBD | TBD | NA | NA | NA | NA |
| 16 | Capture comment text and location, associate with the nearest leaf node | TBD | TBD | NA | NA | NA | NA |
| 18 | Change the Dictionary internal structure so that the columns are held in a hashset, not in a map object | NA | NA | NA | NA | TBD | NA |
| 22 | Handle row selection using “Select top 100” syntax to limit results | TBD | TBD | NA | NA | NA | NA |
| 24 | Concatenations with formula predicands do not parse unless everything is embedded in parentheses; grammar is indeterminate "SELECT substr(strm, 1, 2) || substr(strm, 3, 1) + 1 || substr(strm, 4,1)" | TBD | TBD | NA | NA | NA | NA |
| 25 | Trying to embed single quote in a quated string literal using repetition does not parse “select ‘try embedd’’d quote’” | TBD | TBD | NA | NA | NA | NA |
| 34 | Qualified Joins (e.g., cross, natural, union) do not parse when tuple substitution variables are included | TBD | TBD | NA | NA | NA | TBD |
| 52 | Concatenation doesn’t recognize Predicand Variables | TBD | TBD | NA | NA | NA | TBD |
| 54 | Cross joins and other specialty, “unqualified” join types are not parsed correctly. | TBD | TBD | TBD | NA | NA | TBD |
| 55 | Insert and Update statements use “returning” syntax to provide a cursor of rows inserted or updated. | TBD | TBD | NA | NA | NA | NA |
| 56 | Update and Insert syntax and ASTs, including set statements | TBD | TBD | TBD | TBD | NA | TBD |
| 57 | Add “Select Into Table” syntax variant sometimes used instead of “create table as select” | X | TBD | TBD | NA | TBD | TBD |
| 59 | Add support for cast syntax such as this: “field expression::data type” as a predicand | TBD | TBD | TBD | TBD | NA | NA |
| 60 | Add support for Snowflake’s “IS DISTINCT FROM” conditional operator, and its negative | TBD | TBD | NA | NA | NA | NA |
| 63 | Add support to parse With statements using Inserts and “on conflict” Postgres syntax | TBD | TBD | NA | NA | NA | NA |
| 67 | Parser no longer parses aggregates with embedded calculations: SELECT apple, count(subj + object) from tab1 group by apple | TBD | TBD | NA | NA | NA | NA |
| 68 | Predicands hierarchy is out of whack; too many entry points and several valid scenarios do not parse; recursion is confusing | TBD | TBD | NA | NA | NA | NA |
| 74 | Support Snowflake \_AGG function decorator “within group” with a grouping column | TBD | TBD | NA | NA | NA | NA |
|  | Changes Completed April - June 2019 |  |  |  |  |  |  |
| 23 | Handle row selection using “limit 50” clauses at the end of the query | X | X | NA | NA | NA | NA |
| 53 | Limit statements should parse but are not added to the AST properly | X | X | X | X | NA | NA |
| 58 | Add support for “cast(field as datatype)” casting syntax | X | X | NA | NA | NA | NA |
| 61 | Add Snowflake and Postgres window function options “rows between unbounded preceding and unbounded following” | X | X | NA | NA | NA | NA |
| 62 | Substitution Names accept domain, entity and attribute notations like “<[DOMAIN].[ENTITY].[ATTRIBUTE]>” | X | X | X | X | X | X |
| 64 | Support certain SQL reserved words as column names: DESC, ASC | X | X | X | X | X | NA |
| 65 | Add syntax for all Hive and Snowflake, and many Postgres data Types | X | X | NA | NA | NA | NA |
| 66 | Add support for Snowflake’s “try\_cast” casting function | X | X | NA | NA | NA | NA |
| 69 | Support certain SQL reserved words as column names: RANK | X | X | X | X | X | NA |
| 70 | Support Snowflake window functions FIRST\_VALUE, LAST\_VALUE and NTH\_VALUE | X | X | NA | NA | NA | NA |
| 71 | Support all specialty aggregate functions from Snowflake (August 2019) | X | X | NA | NA | NA | NA |
| 72 | Support Snowflake First\_, Last\_ and nth\_ value decorator syntax: ignore/respect nulls, and select from first or last | X | X | NA | NA | NA | NA |
| 73 | Support Snowflake lead and lag value decorator syntax: ignore/respect nulls, and select from first or last | X | X | NA | NA | NA | NA |
| 75 | Order by decorator (Snowflake) “NULLS LAST” and “NULLS FIRST” | X | X | NA | NA | NA | NA |
|  | Changes Completed Prior to January 2018 |  |  |  |  |  |  |
| 1\* | Parse and create COLUMN substitutions | X | X | X | X | X | X |
| 2\* | Substitution logic can tell the difference between a column substitution and a predicand substitution | NA |  |  |  |  | X |
| 3\* | Parse substitution variables including tables/tuples, join conditions and where clauses |  |  |  |  |  | X |
| 4\* | Modify the ON AST to eliminate the first pair of parentheses, if any | NA | X | NA | NA | NA | NA |
| 5\* | Include condition substitutions from ON, WHERE, CASE and other clauses to the Substitution Variables list | NA | NA | NA | NA | NA | X |
| 6\* | Add column substitution entries to the Interface, Symbol Table, and Table Dictionary | NA | NA | X | X | X | NA |
| 7\* | Include IN LIST, Join Extension substitution variables in the Substitution Variables list | NA | NA | NA | NA | NA | X |
| 8\* | Parse IN conditions with predicand or column substitutions “<col> in (1, 2)” | X | X | X | NA | X | X |
| 9\* | Predicand Parser must add substitutions to the Substitution List and set the proper substitution type (column or predicand) – This needs to be set by knowing the context (location in the grammar) where the variable is encountered. | NA | NA | NA | NA | NA | X |
| 10\* | Add “In List” type of substitution to allow for: “column in <in\_list\_entries>” | X | X | X | NA | NA | X |
| 17\* | Add “optional join extensions” substitution variables. These sit at the end of a join sequence and represent additional join statements. Do we need more than one per join sequence (for instance optional extensions on | X | X | X | NA | X | X |
| 19 | Fix between statement so that it parses and produces a proper subtree | X | X | X | NA | X | X |
| 20\* | Implement “like” clauses; subj\_cd like ‘%STUFF%’ | X | X | X | X | X | NA |
| 21\* | Like clauses with functions (general predicands) defined after the like: “fld like lower(otherField)” and substitution variables “fld like <substitute>” | X | X | X | NA | X | NA |
| 26\* | Window function subtree parses but AST is incomplete; Query Interface includes too many columns (picking up partition parameters, not just its own output); "SELECT lead(code,1) over (partition by spriden\_id order by code)" | X | X | X | X | X | NA |
| 27\* | Case When with substitution variable is not setting the substitution variable type at all. Should be either a predicand or a condition, depending on the case style:  “CASE observation\_time WHEN s948.OBSERVATION\_TM THEN S948.t\_student\_last\_name WHEN <today> THEN…”  Also variable not added to Substitution Variables | X | X | X | X | X | X |
| 28 | Query with tuple substitution in from clause without a table alias will not parse where clause: “from <tablename> where” | Probably OK |  |  |  |  |  |
| 29\* | Query with tuple substitution in from clause with a table alias parses, but scrambled Table Dictionary;  "Select tt.<column1> as redvalue, tt.<column2> as greenvalue  from <table> as tt where tt.<column1> > tt.<column2>;" produces this dictionary: {table\_ref={substitution={name=<column2>, type=column}}} | X | X | X | NA | X | X |
| 30\* | ~~Where~~, ~~Join and Case~~ condition substitution variables are not included in the substitution list | NA | NA | NA | NA | NA | X |
| 31\* | Predicand and column substitutions not included in the Symbol Tree | NA | NA | X | NA | NA | NA |
| 32\* | Predicand and column substitutions included in the Interface as “UNKNOWN” unless they have an alias; Should carry substitution variable name | NA | NA | NA | X | X | NA |
| 33\* | Substitution variables inside of FUNCTIONS are not recognized as predicands, do not get formed properly in the AST, and do not get included in the Substitution List; qualified column substitution variables are okay: “SELECT func(<substitute\_me>, old\_table.<today>, 128.9) as ex” | NA | X | X | NA | X | X |
| 35\* | Interface doesn’t change column case | NA | NA | NA | X | NA | NA |
| 36\* | Sort order by list captures column and predicand variables | X | X | X | NA | X | X |
| 37\* | Group by captures column and predicand variables | X | X | X | NA | X | X |
| 38 | Specialized TRIM function captures column and predicand variables | X | X | X | NA | X | X |
| 39\* | Count Function captures column and predicand variables | X | X | X | NA | X | X |
| 40 | Should predicand variables appear in the symbol table of a query if it is only embedded within another function? | NA | NA | Answer is No | NA | NA | NA |
| 41\* | Like followed by a predicand variable does not parse: col like <variable> | X | X | X | NA | NA | X |
| 42\* | Predicand variable appearing before a Like statement is not recognized or handled | NA | X | X | NA | NA | X |
| 43\* | Where with a single Predicand variable not recognizing type of variable | NA | X | NA | NA | NA | X |
| 44\* | When condition variable in parenthetical condition all by itself, it is not recognized or captured. | NA | X | NA | NA | NA | X |
| 45\* | When a condition variable appears in parentheses within a larger AND/OR clause, it gets labelled as a QUERY variable | NA | X | NA | NA | NA | X |
| 46\* | Condition variable in an ON statement not recognized or classified | NA | X | NA | NA | NA | X |
| 47\* | Parenthetical around ON condition with condition variable where variable is not recognized or classified; Parenthetical should be dropped (see 4) | NA | X | X | NA | X | X |
| 48\* | Predicand variable in an IS NULL condition is not recognized | NA | X | NA | NA | NA | X |
| 49 | Predicand variable in an “is true” comparison doesn’t parse | X | X | NA | NA | NA | X |
| 50\* | Predicand parsing does not create a Table Dictionary, even though a Symbol Table exists | NA | NA | NA | NA | X | NA |
| 51\* | Condition parsing does not create a Table Dictionary, even though a Symbol Table exists | NA | NA | NA | NA | X | NA |

# TO DO LIST

# 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} |
| Trim Function (special variations) | trim(leading '0' from field1) as trimmed | {function={function\_name=trim, parameters={qualifier=leading, trim\_character={literal='0'}, value={column={name=field1, table\_ref=null}}}}, alias=trimmed} |
| 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 | (SELECT aa.scbcrse\_coll\_code FROM scbcrse aa) as INTERNATIONAL\_IND | {lookup={select={1={column={name=scbcrse\_coll\_code, table\_ref=aa}}}, from={table={alias=aa, table=scbcrse}}}, 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 | {between={item={column={name=subj\_code, table\_ref=null}},  symmetry=symmetric, end={column={…}}, begin={column={…}}, operator=not between}}}}} |
| 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 |

## Substitution Variables

A special kind of syntax has been added to the SQL Parse grammar that permits the introduction of “substitution variables” within the body of an otherwise normal query. These substitution variables can be used as placeholders for dynamically-generated SQL engines, and they introduce another level of complexity and power to query handling projects.

With substitution variables, queries can be written to act as templates, specifying the most common logic while providing insertion points for custom or dynamically determined specializations. As a simple example, consider a family of queries used to standardize the tuple output column names over disparate source tables, such as when integrating data from two different systems. In one system a table named “tab1” might have columns “cola, colb” while a second system might have a table named “tabular” with columns named “ex1, ex2”. Using substitution variables, a query such as the following might be defined that could then be resolved at run time to run against either source system:

Select <column1> as redvalue, <column2> as greenvalue from <table> where <column1> > <column2>;

There are several variations of substitution variables, each with a specific type that sets a constraint on the type of AST that can be inserted in that variable.

| Type of Substitutions | SQL Reference | AST Example |
| --- | --- | --- |
| column | Select tt.<StudentId> from tab1 as tt | substitution={name=<StudentId>, type=column} |
| predicand | Select <StudentId> from tab1 where <StudentId> = 100100 | substitution={name=<StudentId>, type=predicand} |
| in\_list | Select val from tab1 where ref in<ListOfValues> | substitution={name=<ListOfValues>, type=in\_list} |
| condition | Select <StudentId> from tab1 where <whereClause> | substitution={name=<whereClause>, type=condition} |
| tuple | Select \* from <StudentTable> | substitution={name=<StudentTable>, type=tuple} |
| query | With getLastXTerms as <GetLastXTerms>, select \* from getLastXTerms | substitution={name=<GetLastXTerms>, type=query} |
| join\_extension | Select \* from tab1 as T join tab2 as P <optionalJoin> | substitution={name=<optionalJoin>, type=join\_extension} |

Substitution variable types are context dependent, and the grammar must be used to set the type since it is not always apparent simply be looking at the immediate context to recognize the type reliably.

## 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={…}}

### assignments

This key points to a list of subtrees representing column assignments in “update” and “insert” subtrees.

EXAMPLE:

assignments={1={…}, 2={…}, 3={…}}

### begin

This key points to a predicand subtree representing the value to be used as the starting range value in a “between” subtree.

EXAMPLE:

begin={column={…}}

### between

This key points to a condition subtree that represents a between statement. There are two instances of the between clause and they will have different subtrees. When the between statement falls in a condition like a where clause, it will have three main subtrees, an operator and possibly an optional symmetry keyword.

It takes a basic form consisting of an “item” predicand to be compared to a range of values, one beginning and one ending the range. It has an “operator” key indicating whether this is a normal between or a “not between”, and can have an option symmetry qualifier.

EXAMPLE:

between={item={…}, symmetry=symmetric, end={column={…}}, begin={column={…}}, operator=not between}}}}}

The second form will only appear as an optional element inside a “bracket”, which is itself an optional part of the “over” subtree in a “window\_function”. In this form, there is no “item”, nor an “operator”, nor a “symmetry” entry. The “begin” and “end” subtrees in this instance will contain a “value” child which will either be a number (representing the number of rows or records with relationship to the current record) or the constant “unbounded” which indicates the frame is bounded (or unbounded) by the number of rows indicated. The second tag in the “begin” and “end” subtrees will be a “direction” indicating whether the given bound value defines the “preceding” or “following” rows relative to the current row. Alternatively, one or the other entries in the “between” could simply be the “value” set to “current row”, which would indicate that the between begins or ends with the current row.

EXAMPLE:

between={end={value=unbounded, direction=FOLLOWING}, begin={value=3, direction=PRECEDING}}

between={end={value=30, direction=PRECEDING}, begin={value=CURRENT ROW}}

between={end={value=unbounded, direction=FOLLOWING}, begin={value=3, direction=PRECEDING}}

### bracket

This key points to a subtree within the “over” subtree of a “window\_function” subtree. This is an optional statement within the over statement that can be used to define a “frame” for the window function which is more limited than the default partition used to define the window for the function. This can be used to set a subset of rows or range within the window to apply the window function against, thereby selecting different values. The bracket defines the set of rows within a partition upon which to apply the function. This could be the entire partition but is more generally used to create sliding frames.

There are two subtree formats, one uses a special “between” subtree, and the other will have a constraint relative to the current record setting a constraint either before or after the current row. Depending on the SQL engine, the default behavior if the frame may be different.

Every bracket subtree will at least have a “type” indicating whether the “rows” of the bracketed frame should be used in the function, or whether the “range” should be used. In the “range” variant, the range indicated is the set of rows with the same values in the partition key. This is a subtle difference, check your engine’s documentation.

After setting the type, the bracket provides the definitional constraints for setting the first row and last row of the frame, relative to the current row. Most typically, this would require a “between” subtree. But in the event the “between” is not present, then a “value” (which could be a number or the constant term “unbounded”, or the constant “current row”). If not the “current row” constant in the “value”, then a “direction” label will also appear relative to the sorted rows in the partition (either “PRECEDING” or “FOLLOWING”).

EXAMPLE:

bracket={type=rows, between={end={value=unbounded, direction=FOLLOWING}, begin={value=unbounded, direction=PRECEDING}}

bracket={type=range, value=30, direction=PRECEDING}

bracket={type=range, value=CURRENT ROW}

### 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. They can be identified by the presense or lack of an “item” subtree in the “case” subtree. Variant 1 does not have an “item” subtree, and variant 2 does.

The first variant contains two immediate children subtrees, one being the “clauses” subtree holding 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. In this variant the “when” clauses are “conditions”, and substitution variables that appear here will be marked accordingly.

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 and the second subtree holding any of the predicands (e.g.,, literal value, column, function result) under the then key.

VARIANT 1 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 the 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). The implied condition, if written out, would appear to be a comparison using equals such that the predicand referenced in the “item” equals the predicand referenced in each “when” (hence, “item”=”when” would not be a bad approximation).

In this variant the “when” clauses are always predicands (of any kind), and substitution variables that appear in this variant’s “when” subtrees will be marked accordingly.

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'}}},

### clauses

This key holds a list subtree containing the “when” and “then” subtrees of a “case” subtree.

EXAMPLE:

clauses={1={then={literal='Y'}, when={literal='a'}}, 2={then={literal='N'}, when={literal='b'}}}

### column

This is a key that holds a subtree containing a column reference. Column references come in three flavors:

1. an actual single column reference,
2. a reference to “all” columns of a table, and
3. a column substitution reference.

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 "\*". This special reference will only appear in the predicand list of a select query. When not qualified, it is a default reference to all of the interfaces from all of the tables, subqueries (or other tuples) in the select statement’s “from” clause.

EXAMPLES:

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

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

COLUMN SUBSTITUTION REFERENCE: A subtree is a column substitution. If it has a table reference (“table\_ref”), then the variable can only be filled by an actual column from the table/query/tuple with the given reference name. If there is no value in the “table\_ref” then that indicates the column can be filled from any column found on any participating tuple, or by any predicand constructed out from any or all of the participating tuples at the same level.

EXAMPLE:

column={ substitution={name=<StudentId>, type=column}, table\_ref=aa}

column={ substitution={name=<StudentAge>, type=column}, table\_ref=null }

### 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}}

### data\_type

This key points to a nested subtree representing a data type definition. There are three kinds of definitions, as follows. “Static” data types are just the name of the data type. “Variable” data types can have an optional length. Finally, “precision” data types can have both precision and scale properties (these are generally numeric data types).

EXAMPLE: data\_type={type=boolean}

data\_type={type=varchar, length=2}

data\_type={type=numeric, precision=9, scale=3}

### dbname

This key points to a leaf node string containing the name of a database. It appears in some variations of the “table” subtree used in the “join” and “from” subtree clauses.

### direction

This key points to a leaf node string containing the direction relative to the current record in a window function’s partition from which to determine the calculation limits to be included in the function. It cane either indicate the relative directions “PRECEDING” or “FOLLOWING”.

direction=PRECEDING

### else

This key points to a predicand value in the “case” subtree. extension={substitution={name=<optionalJoinExtensions>, type=joinExtension}

### end

This key points to a predicand subtree representing the value to be used as the ending range value in a “between” subtree.

EXAMPLE:

end={column={…}}

### extension

This key points to a substitution variable within a “from” subtree containing a “join” subtree.

### 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={…}}

A special substitution variable has been introduced which can form the basis of an optional sequence of additional join statements, if the specific and complete series is unknown or variable.

The purpose of the join extension query is to permit insertion of optional join clauses. This provides the ability to define a core query and then allowing different instances to substitute a variable number of additional constraint joins. This will permit, for example, differences in population queries defined for different use cases.

SUBSTITUTION EXAMPLES:

from tab1 join tab2 on <joinCondition> <optionalJoinExtensions>

from={join={…}, extension={substitution={name=<optionalJoinExtensions>, type=joinExtension}}}}}

### 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.

A third variant supports the “cast” function syntax. In the cast function, the first property is a predicand which is labelled using a “value” key. The function “name” will still be what the user typed in, thus permitting support for various dialect’s versions. However, an extra “type” key will hold the token indicating the specific type of cast function (currently “CAST” and “TRY\_CAST” variations are supported). Finally a “data\_type” key will hold the subtree holding the data type definition to be applied to the resulting value.

A fourth variant supports special decorator syntax on last\_value, first\_value and nth\_value window functions. These decorators are additional instructions to these functions indicating how to handle null values, and whether to select from the start of the window or backwards from the end of the window. A “null\_handle” key with either “ignore” or “respect” values indicates whether to “ignore nulls” or “respect nulls” when selecting the value from the window. A “select\_from” key with either “first” or “last” values indicates whether the “nth\_value” function starts counting values “from first” value or backwards “from last” value in the window.

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={function\_name=cast, type=CAST, value={column={name=scbcrse\_eff\_term, table\_ref=null}}, data\_type={…}}

function={function\_name=Try\_cast, type=TRY\_CAST, value={column={name=scbcrse\_eff\_term, table\_ref=null}}, data\_type={…}}

function={function\_name=last\_value, null\_handle=ignore …}

function={function\_name=nth\_value, null\_handle=respect, select\_from=last …}

### function\_name

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

Example:

function\_name=datestr

### groupby

The “groupby” key holds a list of predicand references forming the “group by” clause of an aggregate query.

Example:

groupby={1={column={…}}, 2={calc{…}}

### having

The “having” key holds any condition subtree forming the “having” condition statement on certain types of queries.

Example:

having={condition={left={function={…}}}, right={literal=201310}, operator=>=}}}

### 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 one of either an “in\_list” subtree or a “not\_in\_list” subtree. Either variation of the “in list” subtree will hold either a list of values, or a subquery that returns a list of values to be searched.

The condition tries to match the predicand “item” value to any or all of the values in the “in list”. For an actual “in\_list” case, the condition is true if the predicand’s value equals one of the entries in the list. For the “not\_in\_list” case, the condition is true if the predicand doesn’t match any of the entries in the list.

STANDARD 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={item={column={name=subj\_code, table\_ref=null}}, not\_in\_list={list={1={literal='AA'}, 2={literal='BB'}}}}

in={item={column={name=subj\_code, table\_ref=null}}, not\_in\_list={query={…}}}}

As in most other places in the AST, the in subtree can hold substitution variable entries. The item could be a predicand substitution variable, or it could hold a column substitution variable. The in\_list could hold a query substitution variable.

But the in subtree itself can also contain a special variation of a substitution variable in the place of the in\_list. A special substitution type has also been defined that can appear in this subtree representing the entire “in\_list” subtree.

This type of in\_list substitution variable can only appear in this context and can only be replaced by a list of literal values.

SUBSTITUTION EXAMPLES:

tab1.<StudentId> in (select …)

in={item={column={ substitution={name=<StudentId>, type=column}, table\_ref=tab1}}, in\_list={query={…}}}}

<StudentId> in (‘AA’, ‘BB’)

in={item={substitution={name=<StudentId>, type=predicand}}, in\_list={list={1={literal='AA'}, 2={literal='BB'}}}}

subj\_code in (<SubjectCodeQuery>)

in={item={column={name=subj\_code, table\_ref=null}}, in\_list={ substitution={name=<SubjectCodeQuery>, type=query}}}

tab1.<StudentId> in <ListOfValues>

in={item={column={ substitution={name=<StudentId>, type=column}, table\_ref=tab1}, substitution={name=<ListOfValues>, type=in\_list})

### 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.

STANDARD EXAMPLES:

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

in\_list={query={…}}}

### insert

This key is used TBD to handle insert SQL statements. It appears TBD.

EXAMPLE:

insert={…} TBD

## 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 intersect subtree. Second, this will point to a subtree which represents the actual intersect statement. The intersect list subtree will have a query subtree for every query participating in the intersection, and will also include a child intersect subtree for every intersect clause.

The intersect 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 intersect statement ordering, this will typically appear as an alternating list of a query subtree then intersect statement subtree, then query subtree.

The intersect 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 intersect. Optionally, this subtree will include a qualifier key which is either null or indicates the intersect “all” option.

EXAMPLES:

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

intersect ={qualifier=null, operator= intersect }

### 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. |

STANDARD 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}}

### length

Key points to a leaf node key containing the length property of a variable data type.

EXAMPLE: length=10

### limit

This key points to a subtree containing the number of rows to be returned by the query.

EXAMPLE: limit={literal=100}

### list

This key appears within an “in\_list” subtree and contains the list of actual, literal values that should be checked by the “in” condition.

STANDARD EXAMPLES:

list={1={literal='AA'}, 2={literal='BB'}}

### literal

This key holds a leaf node string containing the literal text copied directly from the SQL statement. This could include quotes, if the original literal value was quoted, or just numbers.

EXAMPLES:

literal=100

literal='AA'

### lookup

This key holds an embedded subquery subtree when that subquery appears as a predicand in the query statement. It can also hold a “query” substitution variable.

EXAMPLES:

lookup={query={…}}

### 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={condition={left={column={…}}, right={column={…}}, operator==}}}

### not\_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.

This is the negative version of the “in\_list” subtree and is used to hold the list of all of values that should NOT match the test value.

STANDARD EXAMPLES:

not\_in\_list={list={1={literal='AA'}, 2={literal='BB'}}}

not\_in\_list={query={…}}}

### null\_handle

A subkey that directs last\_value, first\_value and nth\_value window functions how to consider null values in their processing. Valid values are “respect” to respect null values in the selection, and “ignore” which makes the function skips null values. The SQL output should be “ignore nulls” or “respect nulls”.

Example:

null\_handle=ignore

null\_handle=respect

### 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 actually 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 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.

The original query may or may not have enclosed the on statement in parentheses. When parsed, the AST generated will remove the outermost pair of parentheses from the on subtree. Hence “on (<condition>)” and “on <condition>” will both produce the same AST.

EXAMPLE:

on={condition={left={column={…}}, right={column={…}}, operator==}}}}

on={substitution={name=<studentPopulationJoinCondition>, type=condition}}

### 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, predicand={column={…}}, null\_order=null}, 2={sort\_order=desc, predicand={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={…}}

## parts

This key is an optional subtree of a substitution variable tree. If the substitution variable name follows the internal-structure convention, then the parts subtree separates the parts of internal structure. It can hold from one to three entries, simply ordered numerically.

EXAMPLES:

parts={1=[ENTITY], 2=[ATTRIBUTE]}

parts={1=[DOMAIN], 2=[ENTITY]}

parts={1=[DOMAIN], 2=[ENTITY], 3=[ATTRIBUTE]}

parts={1=[ATTRIBUTE]}

### precision

Key points to a leaf node key containing the precision property of a numeric data type.

EXAMPLE: precision=10

## predicand

This key is part of an “order\_by” subtree and holds the predicand subtree, of whatever type.

EXAMPLES:

predicand={column={…}}

predicand={calc={…}}

predicand={substitution={…}}

### 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 the special, database-specific (e.g., Postgres) version of a “trim” statement, possibly others.

EXAMPLES:

qualifier=distinct

qualifier=all

### query

This key is used within the “SQL” top key tree when the query is constructed from a set of “with” queries. In this circumstance, the “query” key points to the ultimate query statement which would use the named queries from the “with” subtree.

EXAMPLE:

query={select={…}}

### returning

This key is used as a subtree within “insert” and “update” statements. TBD

EXAMPLE: returning={… }

### 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 statement 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={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={…}}}}

### scale

Key points to a leaf node key containing the scale property of a numeric data type.

EXAMPLE: scale=10

### 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={...}}}

### select\_from

A subkey that directs nth\_value window functions from which end of the window to start searching for its value. Valid values are “first” to start at the lowest value in the ordered window and count n values to select its result, or “last” to start at the final value in the ordered window and count n values backwards to select its result. The SQL output should be “from last” or “from first”.

Example:

select\_from=first

select\_from=last

### set

The key contains a subtree representing an assignment of value in an update statement.

EXAMPLE: set={…} TBD

### 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

### symmetry

The key is used as a leaf node reference inside of a “between” subtree. It holds the token representing the type of symmetry to be applied when assessing the range condition.

EXAMPLE: symmetry = symmetric

### substitution

The substitution key is a leaf node that holds a placeholder for an unspecified substitution into the parsed tree. These are special symbols which can act as variables or macro-insertion points, when used in a substitution context. Within the unparsed query, substitution variables appear within angle brackets, namely these “<>”.

In some situations, the query being parsed will not specify certain details of a finished, executable query, but will act as a template for a family of similar queries. The substitution entries in the AST will be places that allow insertions of arbitrary extensions (in the form of new AST subtrees) and must be filled before the query can be executed.

Substitutions are typed based on their location within the primary AST. They can represent four different things. The simplest substitution is a predicand, like an actual column reference, a formula, or any other of the predicand types listed at the beginning of this document. Other types include conditions, such as appear in where clauses or joins, tuples which may be subqueries or merely a physical table reference as in a from or join sequence, and queries which could only appear where a query would be defined, as in a With statement.

Predicand Substitutions can appear anywhere a predicand can appear. Likewise, condition Substitutions can appear wherever a condition would appear, and tuple Substitutions can appear where tables or subqueries might appear.

Predicand variables may only be the actual predicand, and not the alias or result name. Hence there is no valid construct such as <table>.<column>, since in this context, the <table> is actually the alias of a table or subquery.

When encountered, the substitution can be replaced by any new AST subtree satisfying the required type (predicand, condition or tuple). Some substitutions will have an effect on the final Symbol Tree and other artifacts generated by the SQL Walker.

Special logic has been added to permit additional structure to be embedded in the variable name using bracket identifiers separated by periods. This syntax could be used for several extensions, the example shown uses it to represent a logical data model reference consisting of a “domain”, an “entity type” from the domain, and an “attribute” of that entity. These extra elements are pushed under an optional subtree under the “parts” label. Up to three dot-separated name parts can be included using this syntax, so long as each part is surrounded by square brackets.

EXAMPLE:

substitution={name=<StudentId>, type=column}

substitution={name=<StudentId>, type=predicand}

substitution={name=<whereClause>, type=condition}

substitution={name=<StudentTable>, type=tuple}

substitution={name=<GetLastXTerms>, type=query}

substitution={name=<InListItems>, type=in\_list}

substitution={name=<OptionalJoinExtension>, type=join\_extension}

substitution={name=<[DOMAIN].[ENTITY].[ATTRIBUTE]>, parts={1=[DOMAIN], 2=[ENTITY], 3=[ATTRIBUTE]}, type=column}

### 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}

### then

This key points to a predicand result value in a “when-then” pair of a “clauses” list entry within a “case” subtree.

### to

The key contains a the predicand subtree to be used when setting the value of a target column in an update statement.

EXAMPLE: to={…} TBD

### trim\_character

The key appears as a special extra subtree within a “function” subtree when that function is one of the alternative versions of the “trim” function (e.g., the Postgres Trim function takes special named parameters).

EXAMPLE: trim\_character={literal='0'}

### type

The leaf-node key appears in multiple other complex subtrees and holds a typifier constant appropriate for each context.

### 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}

value={column={name=field1, table\_ref=null}}

### update

This key is used TBD to handle update SQL statements. It appears TBD.

EXAMPLE:

update={…} TBD

### value

The key appears as a special extra subtree within a “function” subtree when that function is one of the alternative versions of the “trim” function (e.g., the Postgres Trim function takes special named parameters).

EXAMPLES:

value={column={name=field1, table\_ref=null}}

### when

Depending on the variant of the “case” subtree that this key appears in, it will point either to a “condition” subtree (for Variant 1) or a predicand comparison value (for Variant 2) in a “when-then” pair of a “clauses” list entry within a “case” subtree.

### where

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

EXAMPLES:

where={and={…}}

where={or={…}}

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

### 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 as documented in the following table.

| Window Function | Usage |
| --- | --- |
| Avg | Calculates the average value across a partition of the data |
| First\_Value | Takes the first value from a set that’s been ordered by the window statement. Can have an optional window frame further limiting the potential set of rows from which the value can selected, |
| Lag | Takes the value from the prior row in a sorted list of rows within a partition |
| Last\_Value | Takes the last value from the last row from a set that’s been ordered. Can have an optional window frame further limiting the potential set of rows from which the value can selected, |
| Lead | Takes the value from the next row in a sorted list of rows within a partition |
| Max | Calculates the maximum value of a column across a partition window on a larger set |
| Min | Calculates the minimum value of a column across a partition window on a larger set |
| *Nth\_Value* | *Selects the value on the row N rows away from the current row. Combined with a window frame statement this function takes two parameters.* |
| Sum | Calculates the summation value of a column across a partition window on a larger set |
| Count | Counts the total number of rows within a partition window on a larger set of rows |
| Rank | Calculates the rank number (sequential position of each row) within a partitioned window |
| Row\_Number |  |
| STDDEV\_POP | Calculates the standard deviation value of a column across the complete population of rows within a partition window on a larger set |
| STDDEV\_SAMP | Calculates the standard deviation value of a column across a randomized sample of rows within a partition window on a larger set |
| VAR\_\_POP | Calculates the variance value of a column across the complete population of rows within a partition window on a larger set |
| VAR\_SAMP | Calculates the variance value of a column across a randomized sample of rows within a partition window on a larger set |

Some SQL engines support additional syntax providing cumulative and sliding frames within a window (such as SQL Server and Snowflake). The combinations of window functions and sliding frames, and the exact behavior implemented for certain syntactic constructions may NOT be the same from one engine to another. The grammar will allow parsing of all variations, even if certain target engines cannot execute the specific combination. The SQL developer is referred to the documentation for their specific engine for clarifications, and will need to handle error prevention for these separately.

EXAMPLES: Standard Window Function Without Frames

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}}

EXAMPLES: Window Functions With Frames

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=nth\_value, 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 appear as a sibling entry at the same level as the with statement 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 several types of AST Variables defined, each substituting for a different portion of a tree. These are as follows:

Predicand Value Variables

Entity Type Variables

Attribute Type Variables

Substitution Variables

Population Variables

### Predicand Value Variables

These variables are used to hold one or more predicand values, of any type. These can appear anywhere a predicand can appear, such as columns in a select statement, or as comparison values in any of the conditions. There are two kinds of these, the first being “independent” predicands which hold values that are universally referenced, regardless of the context of the query. As examples, both a “system time of day” and any of several kinds of contextual references such as “member identity” would be considered to be “independent” of the small context defined by the various levels of a SQL statement. Independent predicand variables are recognized by having two leading “#” signs, a variable name, and a trailing “#” sign, for example: “##system time#”.

The second type of predicand variable, therefore, would be a “dependent” predicand. These variables are recognized by having a single “#” sign to start, the variable name then a trailing “#”. These are more restricted in their usage, because they must be associated within a specific context of the query, namely, they will be associated with one of the underlying tables of the query. As an example, consider a condition using comparing a table column to a constant value or set of values: “course.level\_cd = 50”. To make this a more generic statement, a dependent predicand variable could be used as a substitute: “course.level\_cd = #undergraduate#”. In this example, the values that would be used to fill the #undergraduate# variable must be compatible with the contents of the level\_cd column of the course table.

Often, these variables will be filled with literal values, but any stand-alone predicand is acceptable, so long as it satisfies the context (dependent or independent). This could be literals, column names, formulas, functions, and even lookup subqueries.

### Entity Type Variables

These variables are 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]”.

Only

### Attribute Variables

These variables are 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.

### Substitution Variables

These variables are used to hold an arbitrary substitution point for various subsections of a SQL abstract syntax tree. These have been described earlier and are the primary, and most general substitution types recognized. These are recognized by the angle brackets and are dynamically typed based on where they appear in the properly constructed SQL statement. See other sections for details.

### Population Variables

These variables are 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}. These are probably obsolete and duplicative to the more general substitution variables, and are included here in case their more specialized usage becomes important at a later time.

# Symbol Table

The SQL Parse Event Walker tries very hard to associate columns to their tables. It is not always possible, due to SQL’s ability to implicitly reference columns to be resolved only at run time. In this way, SQL can be understood to be an imprecise, loosely typed language because of the ambiguity it is able to work under. SQL requires, in fact, that it be interpreted within a context that has more information than is exposed in a single SQL statement.

In the absence of that context, it is not always possible to know definitively how column references are associated to their tables. This uncertainty has a direct effect on the organization and accuracy of both the Symbol Table and the Table Dictionary that this utility produces.

The Symbol Table is not really a table, but a hierarchical structure that captures the layers of contexts that are created by the operators and constructs of SQL for combining the data of tables and other queries. The Symbol Table captures the “symbols” of the query as a nested Map<> object. Each level of the Symbol Table represents the symbols defined and used within a specific query or subquery within the statement.

## Structure and Interpretation

Every subquery in the statement is given a symbol table of its own, labeled with the type of statement it is and a unique number. These identifiers are arbitrary, in that the specific symbol table reference given to any particular subquery or construct is determined by how many there are and the order in which they are encountered.

Each Symbol Table query contains three kinds of entries, as follows.

* A subtree containing the outward facing “interface” of the query. These are the results of the subquery, in particular the “aliases” of the columns and expressions generated by the query.
* Subtrees for each actual table (or subquery) containing the names of the columns of the actual table (or the names of the interface of the subquery).
* Entries associating the table or subquery to its alias within the subquery.

In addition, when there is uncertainty about which table or subquery owns a column, an extra subtree will be included labeled as “unknown” where all of the ambiguous column references are stored.

## Examples

The following shows examples of what the Symbol Table structure and content would be for various kinds of queries. Each of the queries establishes a block level context within which a set of tables and columns names are defined. Both tables and columns can be aliased, and so in this presentation, the queries and embedded tables are color coded to reflect their relationship.

| SQL Statement | Symbol Table | Notes |
| --- | --- | --- |
| Basic Select Queries | | |
| SELECT 🡨 **query0**  aa.scbcrse\_coll\_code, 🡨 **interface entries**  aa.\*  FROM  scbcrse 🡨 **actual table** as aa, 🡨 **table alias**  mycrse as courses  WHERE  not aa.scbcrse\_subj\_code = courses.subj\_code  AND (aa.scbcrse\_crse\_numb = courses.crse\_numb  or aa.scbcrse\_crse\_numb = courses.crse\_numb) | {  query0={  **table alias** 🡪 aa=scbcrse,  courses=mycrse,  **actual table** 🡪 scbcrse={  scbcrse\_coll\_code=[@1,7:8='aa',<205>,1:7],  scbcrse\_crse\_numb=[@35,175:176='aa',<205>,1:175],  \*=[@5,29:30='aa',<205>,1:29],  scbcrse\_subj\_code=[@18,83:84='aa',<205>,1:83]},  mycrse={  subj\_code=[@22,106:112='courses',<205>,1:106],  crse\_numb=[@39,198:204='courses',<205>,1:198]},  **interface entries** 🡪 interface={  scbcrse\_coll\_code={column={name=scbcrse\_coll\_code,  table\_ref=aa}},  \*={column={name=\*, table\_ref=aa}}}}  } | The example is a single query. Therefore there is only one query’s symbol table. Every subquery in the statement is given a symbol table of its own, labeled with the type of statement it is and a unique number. Hence in this example, with a basic select query, the label “query0” is assigned to it. |
| select 🡨 **query0**  rec\_type as RECORD\_TYPE,  action\_cd as ACTION,  tag\_name as TAG,  grp\_id as GROUP\_ID,  user\_id as PRIMARY\_USER\_ID  from  tagTbl  where  tag\_name is not null  and length(trim(tag\_name)) > 0  and grp\_id is not null  and length(trim(grp\_id)) > 0  and user\_id is not null  and length(trim(user\_id)) > 0 | {  query0={  interface={  PRIMARY\_USER\_ID={column={name=user\_id,  table\_ref=null}},  ACTION={column={name=action\_cd, table\_ref=null}},  RECORD\_TYPE={column={name=rec\_type,  table\_ref=null}},  TAG={column={name=tag\_name, table\_ref=null}},  GROUP\_ID={column={name=grp\_id, table\_ref=null}}},  tagTbl={  grp\_id=[@47,228:233='grp\_id',<205>,1:228],  action\_cd=[@5,33:41='action\_cd',<205>,1:33],  rec\_type=[@1,8:15='rec\_type',<205>,1:8],  user\_id=[@62,281:287='user\_id',<205>,1:281],  tag\_name=[@32,174:181='tag\_name',<205>,1:174]}}  } | The main difference between this example and the prior example is that in this example, all of the output columns have been aliased. Thus, the Interface contains the aliases of each column as the column key, whereas, in the prior example, the interface used the column names as they were, since no aliases were defined in the query. |
| SELECT 🡨 **query0**  scbcrse\_subj\_code as subj\_code,  count(\*) as total, 🡨 **function**  MAX(scbcrse\_eff\_term) as maximum🡨 **function**  FROM scbcrse  group by scbcrse\_subj\_code  order by 2, scbcrse\_subj\_code, 1 | {  query0={  scbcrse={  scbcrse\_subj\_code=[…],  scbcrse\_eff\_term=[…]},  interface={  total={function={function\_name=COUNT, 🡨 **function**  qualifier=null,  parameters=\*}},  subj\_code={column={name=scbcrse\_subj\_code,  table\_ref=null}},  maximum={function={function\_name=MAX,🡨 **function**  qualifier=null,  parameters={column={  name=scbcrse\_eff\_term,  table\_ref=null}}}}  }}  } | These next two examples show how the Symbol Table represents literal constants, functions, expressions or formulas that are aliased.  The column entries in the Interface in each case will use the alias as the key with the predicand subtree as the value. Hence, in the example, the value for the “total” key is the “count” function. |
| SELECT 🡨 **query0**  scbcrse\_subj\_code as subj\_code,  count(\*),🡨 **unnamed0**  MAX(scbcrse\_eff\_term) 🡨 **unnamed1**  FROM scbcrse  group by scbcrse\_subj\_code  order by 2, scbcrse\_subj\_code, 1 | {  query0={  scbcrse={  scbcrse\_subj\_code=[…],  scbcrse\_eff\_term=[…]},  interface={  subj\_code={column={name=scbcrse\_subj\_code,  table\_ref=null}},  unnamed0={function=  {function\_name=COUNT,🡨 **unnamed0**  qualifier=null,  parameters={column={  name=\*,  table\_ref=null}}}}  unnamed1={function={function\_name=MAX,🡨 **unnamed1**  qualifier=null,  parameters={column={  name=scbcrse\_eff\_term,  table\_ref=null}}}}  }}  } | This is the same example as the previous one, except that the aggregate functions do not have aliases. Depending on the DBMS used to execute such a query, the resulting column will be given a generated name by the DBMS at run time. For our purposes, we don’t know what the name will be, so we assign our own generated name for each unique column output. The SQL Parse Event Walker will use the term “unnamed” and then a sequence number as the resulting column alias for the Interface. |
| Queries Inside of Queries | | |
| SELECT first FROM third  union select second from fifth  union select fourth from sixth  union select seventh from eighth | {  union4={  query0={  third={  first=[@1,8:12='first',<77>,1:8]},  interface={first={column={name=first, table\_ref=null}}}  },  interface={first=query\_column},  query1={  fifth={  second=[@6,39:43='third',<205>,1:39]},  interface={second={column={name=second,  table\_ref=null}}}},  query2={  sixth={  fourth=[@11,70:75='fourth',<205>,1:70]},  interface={fourth={column={name=fourth,  table\_ref=null}}}},  query3={  eighth={  seventh=[@16,102:108='seventh',<205>,1:102]},  interface={seventh={column={name=seventh,  table\_ref=null}}}}  }  }  {  sixth={fourth=[@11,70:75='fourth',<205>,1:70]},  third={first=[@1,8:12='first',<77>,1:8]},  eighth={seventh=[@16,102:108='seventh',<205>,1:102]},  fifth={second=[@6,39:43='second',<205>,1:39]}  } |  |
| SELECT first FROM third  intersect select second from fifth  intersect select fourth from sixth  intersect select seventh from eighth | {  intersect4={  query0={  third={  first=[@1,8:12='first',<77>,1:8]},  interface={first={column={name=first, table\_ref=null}}}  },  interface={first=query\_column},  query1={  fifth={  second=[@6,39:43='third',<205>,1:39]},  interface={second={column={name=second,  table\_ref=null}}}},  query2={  sixth={  fourth=[@11,70:75='fourth',<205>,1:70]},  interface={fourth={column={name=fourth,  table\_ref=null}}}},  query3={  eighth={  seventh=[@16,102:108='seventh',<205>,1:102]},  interface={seventh={column={name=seventh,  table\_ref=null}}}}  }  }  {  sixth={fourth=[@11,70:75='fourth',<205>,1:70]},  third={first=[@1,8:12='first',<77>,1:8]},  eighth={seventh=[@16,102:108='seventh',<205>,1:102]},  fifth={second=[@6,39:43='second',<205>,1:39]}  } | Structurally, this query and the one previous will produce exactly the same Symbol Table and Table Dictionary. This is because the Union and Intersect operations affect these structures in exactly the same way. The only difference is the outermost reference label which indicates how the symbols are combined. |
| SELECT aa.scbcrse\_coll\_code FROM scbcrse aa  WHERE aa.scbcrse\_subj\_code = courses.subj\_code  AND aa.scbcrse\_crse\_numb = courses.crse\_numb  AND aa.scbcrse\_eff\_term =  (SELECT MAX(scbcrse\_eff\_term)  FROM scbcrse  WHERE scbcrse\_subj\_code = courses.subj\_code  AND scbcrse\_crse\_numb = courses.crse\_numb  AND scbcrse\_eff\_term <= courses.term) | {  query1={  aa=scbcrse,  courses={  subj\_code=[@12,74:80='courses',<205>,1:74],  crse\_numb=[@20,120:126='courses',<205>,1:120]},  scbcrse={  scbcrse\_coll\_code=[@1,7:8='aa',<205>,1:7],  scbcrse\_crse\_numb=[@16,97:98='aa',<205>,1:97],  scbcrse\_eff\_term=[@24,143:144='aa',<205>,1:143],  scbcrse\_subj\_code=[@8,51:52='aa',<205>,1:51]},  interface={  scbcrse\_coll\_code={column={name=scbcrse\_coll\_code,  table\_ref=aa}}},  query0={  courses={  subj\_code=[@39,238:244='courses',<205>,1:238],  term=[@51,324:330='courses',<205>,1:324],  crse\_numb=[@45,281:287='courses',<205>,1:281]},  scbcrse={},  interface={  unnamed={function={function\_name=MAX,  qualifier=null,  parameters={  column={name=scbcrse\_eff\_term,  table\_ref=null}}}}  },  unknown={  scbcrse\_crse\_numb=[…],  scbcrse\_subj\_code=[…],  scbcrse\_eff\_term=[…]}  }  }  }  {  courses={  term=[@51,324:330='courses',<205>,1:324],  crse\_numb=[@20,120:126='courses',<205>,1:120],  subj\_code=[@12,74:80='courses',<205>,1:74]},  scbcrse={  scbcrse\_coll\_code=[@1,7:8='aa',<205>,1:7],  scbcrse\_crse\_numb=[@16,97:98='aa',<205>,1:97],  scbcrse\_eff\_term=[@24,143:144='aa',<205>,1:143],  scbcrse\_subj\_code=[@8,51:52='aa',<205>,1:51]}  } | An example of a subquery embedded as a predicand in a Where condition. The definition of the subquery’s Symbol Table appears at the same level as other tables in the main query. The fact that the query is part of a condition statement is not apparent in the Symbol Table. When interpreting this table, actual tables refer to their columns directly, while the columns available from the subquery must be retrieved from their “interface” subtrees.  NOTE: In this query, the fact that the “courses” table reference is not defined would cause a run-time error. This illustrates how something can be grammatically correct, but unusable. In order to trap this, a validation after the fact would need to look for table references that are not defined and raise a secondary error. This may be added later as an enhancement or secondary function of the SQL Parse Event Walker. |
| SELECT first\_item,  (SELECT item  FROM sgbstdn  WHERE sgbstdn\_levl\_code = 'US') AS INTERNATIONAL\_IND  FROM sgbstdn | {  query1={  sgbstdn={  first\_item=[@1,8:17='first\_item',<205>,1:8]},  interface={  first\_item={column={name=first\_item, table\_ref=null}},  INTERNATIONAL\_IND={lookup={  select={1={column={name=item, table\_ref=null}}},  from={table={alias=null, table=sgbstdn}},  where={condition={  left={column={name=sgbstdn\_levl\_code,  table\_ref=null}}, right={literal='US'}, operator==}}}}},  query0={  sgbstdn={  sgbstdn\_levl\_code=[…],  item=[…]},  interface={  item={column={name=item, table\_ref=null}}}  }  }  }  {  sgbstdn={  sgbstdn\_levl\_code=[...],  item=[…],  first\_item=[…]}  } | Another example of a subquery used as a predicand. As in the prior example, note that the Symbol Table treats the subquery’s Symbol Table as a disconnected reference.  <? This looks odd, now that I look at it critically. Perhaps the reference to the subquery should be inserted under the “lookup” reference instead of including the entire AST statement. ?> |
| SELECT 🡨 **query1**  b.att1,  b.att2  from  (SELECT 🡨 **query0**  a.col1 as att1,  a.col2 as att2  FROM tab1 as a  WHERE a.col1 <> a.col3) AS b | {query1={  b=query0,  def\_query0={  a=tab1,  tab1={  col2=[@17,53:53='a',<205>,1:53],  col3=[@31,100:100='a',<205>,1:100],  col1=[@27,90:90='a',<205>,1:90]},  interface={  att2={column={name=col2, table\_ref=a}},  att1={column={name=col1, table\_ref=a}}  }},  interface={  att2={column={name=att2, table\_ref=b}},  att1={column={name=att1, table\_ref=b}}},  query0={  att2=[@5,16:16='b',<205>,1:16],  att1=[@1,8:8='b',<205>,1:8]}  }}  {tab1={  col2=[@17,53:53='a',<205>,1:53],  col3=[@31,100:100='a',<205>,1:100],  col1=[@27,90:90='a',<205>,1:90]}} | When the embedded subquery is used in the From or Join statements, then it must be treated differently than when it is used in other contexts (as in the prior examples, as a predicand). This is so that the data type can disambiguate between the role the query plays in the context of the query it is embedded in (query0), and its own internal construction and symbol table (def\_query0). Within the context of query1, query0 acts as just another table or relation, with a set of contributing columns referenced and handled in the normal way. However, in its own right, it has an embedded Symbol Table which defines it. Note, however, that the Table Dictionary correctly understands that the query0 does not represent an external relationship, hence it only contains the references from table tab1. |
| Insert Queries | | |
| insert into  employees  (emp\_sales\_count,  redder)  values  (select  acct\_sales\_count + 1,  greener  FROM accounts) | {  employees={  emp\_sales\_count=[…],  redder=[…]},  query0={  accounts={  acct\_sales\_count=[…],  greener=[...]},  interface={  greener={column={name=greener, table\_ref=null}},  unnamed={calc={  left={column={name=acct\_sales\_count,  table\_ref=null}},  right={literal=1},  operator=+}}  }  }  }  {accounts={  acct\_sales\_count=[…],  greener=[...]},  employees={  emp\_sales\_count=[…],  redder=[…]}} |  |
| Update Queries | | |
| update  this\_table  set outputA = column1,  outputB = column2,  outputC = column3  from that\_table  where  this\_table.key=that\_table.key | {  that\_table={  key=[@21,116:125='that\_table',<205>,1:116]},  this\_table={  outputC=[@11,60:66='outputC',<205>,1:60],  outputA=[@3,22:28='outputA',<205>,1:22],  outputB=[@7,41:47='outputB',<205>,1:41],  key=[@17,101:110='this\_table',<205>,1:101]},  unknown={  column1=[@5,32:38='column1',<205>,1:32],  column3=[@13,70:76='column3',<205>,1:70],  column2=[@9,51:57='column2',<205>,1:51]}  }  {  that\_table={  key=[…]},  this\_table={  outputC=[...],  outputA=[@3,22:28='outputA',<205>,1:22],  outputB=[@7,41:47='outputB',<205>,1:41],  key=[@17,101:110='this\_table',<205>,1:101]}  } | <? These unknowns should obviously have been put into the “that\_table” dictionary and symbol table, since tat is the only place they could have come from. Need to add “unknown” cleanup logic to the update query step… ?> |
| With Queries | | |
| WITH upsert AS  (Select  concentration\_desc,  stvmajr\_desc  FROM  bnr\_stvmajr)  Select  cat\_concentration,  concentration\_code,  concentration\_desc,  active\_ind  FROM upsert | {  with={  upsert={  query0={  interface={  concentration\_desc={  column={name=concentration\_desc,  table\_ref=null}},  stvmajr\_desc={  column={name=stvmajr\_desc, table\_ref=null}}  },  bnr\_stvmajr={  concentration\_desc=[...],  stvmajr\_desc=[…]}  }  }  },  query1={  upsert={  concentration\_desc=[…],  cat\_concentration=[…],  concentration\_code=[…],  active\_ind=[…]},  interface={  concentration\_desc={column={name=concentration\_desc,  table\_ref=null}},  cat\_concentration={column={name=cat\_concentration,  table\_ref=null}},  concentration\_code={column={name=concentration\_code,  table\_ref=null}},  active\_ind={column={name=active\_ind, table\_ref=null}}  }  }  }  {  upsert={  concentration\_desc=[…],  cat\_concentration=[…],  concentration\_code=[…],  active\_ind=[…]},  bnr\_stvmajr={  concentration\_desc=[...],  stvmajr\_desc=[…]}} |  |
| WITH  upsert AS  (UPDATE cat\_concentration  SET concentration\_desc = stvmajr\_desc  FROM bnr\_stvmajr  RETURNING \* )  INSERT INTO cat\_concentration  ( concentration\_code,  concentration\_desc,  active\_ind)  values (  SELECT  stvmajr\_code AS concentration\_code,  stvmajr\_desc AS concentration\_desc,  'T' AS active\_ind  FROM  bnr\_stvmajr  WHERE NOT EXISTS (  SELECT \*  FROM upsert)  AND stvmajr\_valid\_concentratn\_ind = 'Y') | {with={upsert={cat\_concentration={concentration\_desc=[@7,48:65='concentration\_desc',<205>,1:48]}, bnr\_stvmajr={}, unknown={stvmajr\_desc=[@9,69:80='stvmajr\_desc',<205>,1:69]}}}, cat\_concentration={}, query2={interface={concentration\_desc={column={name=stvmajr\_desc, table\_ref=null}}, concentration\_code={column={name=stvmajr\_code, table\_ref=null}}, active\_ind={literal='T'}}, bnr\_stvmajr={stvmajr\_desc=[@32,266:277='stvmajr\_desc',<205>,1:266], stvmajr\_valid\_concentratn\_ind=[@51,392:420='stvmajr\_valid\_concentratn\_ind',<205>,1:392], stvmajr\_code=[@28,228:239='stvmajr\_code',<205>,1:228]}, query1={upsert={\*=[@46,368:368='\*',<195>,1:368]}, interface={\*={column={name=\*, table\_ref=\*}}}}}, unknown={concentration\_desc=[@21,177:194='concentration\_desc',<205>,1:177], concentration\_code=[@19,156:173='concentration\_code',<205>,1:156], active\_ind=[@23,198:207='active\_ind',<205>,1:198]}}  {with={upsert={cat\_concentration={concentration\_desc=[@7,48:65='concentration\_desc',<205>,1:48]}, bnr\_stvmajr={}, unknown={stvmajr\_desc=[@9,69:80='stvmajr\_desc',<205>,1:69]}}}, cat\_concentration={concentration\_desc=[@7,48:65='concentration\_desc',<205>,1:48]}, upsert={\*=[@46,368:368='\*',<195>,1:368]}, bnr\_stvmajr={stvmajr\_desc=[@32,266:277='stvmajr\_desc',<205>,1:266], stvmajr\_valid\_concentratn\_ind=[@51,392:420='stvmajr\_valid\_concentratn\_ind',<205>,1:392], stvmajr\_code=[@28,228:239='stvmajr\_code',<205>,1:228]}} |  |

# Table Dictionary

The table dictionary is a collection of the tables and columns referenced within a query. The SQL Parse Event Walker tries to associate each column with the correct table using clues it derives from the query. The most obvious examples are when the query has table references prefixing the column names, or if the query only has one table.

The dictionary derives the relationship between tables and columns by assuming that the query is correctly written. In the absence of the contextual information that the DBMS has when actually running and checking the validity of a query against its own, internal dictionary, the only information available to the SQL Parse Event Walker is the actual query itself. Given a corpus of SQL queries, this capability can be used to derive a model of the database directly from the queries.

The contents of the dictionary object consist of only the objects recognized as actual tables and their columns within a query. No matter how layered or complex the input query (e.g., no matter how many subqueries, joins, unions, intersects, formulas, conditions, etc.), the Table Dictionary will only ever produce the set of actual tables and columns making up the inputs to the query.

<? TODO: Change the Dictionary internal structure so that the columns are held in a hashset, not in a map object ?>

| SQL Statement | Table Dictionary | Notes |
| --- | --- | --- |
| Basic Select Queries | | |
| SELECT  aa.scbcrse\_coll\_code,  aa.\*  FROM  scbcrse as aa,  mycrse as courses  WHERE  not aa.scbcrse\_subj\_code = courses.subj\_code  AND (aa.scbcrse\_crse\_numb = courses.crse\_numb  or aa.scbcrse\_crse\_numb = courses.crse\_numb) | {scbcrse=  {scbcrse\_coll\_code=[@1,7:8='aa',<205>,1:7],  scbcrse\_crse\_numb=[@35,175:176='aa',<205>,1:175],  \*=[@5,29:30='aa',<205>,1:29],  scbcrse\_subj\_code=[@18,83:84='aa',<205>,1:83]},  mycrse =  { subj\_code =[@22,106:112='courses',<205>,1:106],  crse\_numb =[@39,198:204='courses',<205>,1:198]}  } | In this query, there are only two tables, and every column is associated, either directly or through a table reference alias, to either one table or the other. Hence the Table Dictionary shows every column in its own table. |
| select  rec\_type as RECORD\_TYPE,  action\_cd as ACTION,  tag\_name as TAG,  grp\_id as GROUP\_ID,  user\_id as PRIMARY\_USER\_ID  from  tagTbl  where  tag\_name is not null  and length(trim(tag\_name)) > 0  and grp\_id is not null  and length(trim(grp\_id)) > 0  and user\_id is not null  and length(trim(user\_id)) > 0 | {tagtbl=  {grp\_id=[@47,228:233='grp\_id',<205>,1:228],  action\_cd=[@5,33:41='action\_cd',<205>,1:33],  rec\_type=[@1,8:15='rec\_type',<205>,1:8],  user\_id=[@62,281:287='user\_id',<205>,1:281],  tag\_name=[@32,174:181='tag\_name',<205>,1:174]}  } | Since there is only one table in the query, it is obvious that all columns referenced must be in that table. Hence the Table Dictionary shows all columns inside it. |
| SELECT  scbcrse\_subj\_code as subj\_code,  count(\*) as total,  MAX(scbcrse\_eff\_term) as maximum  FROM scbcrse  group by scbcrse\_subj\_code  order by 2, scbcrse\_subj\_code | {scbcrse=  {scbcrse\_subj\_code=[@...],  scbcrse\_eff\_term=[@...]}  } | The next two examples show an important detail. Namely, it does not matter what the aliases are or even if there are aliases defined for actual columns of the input when constructing the Tale Dictionary of otherwise similar queries. |
| SELECT  a.scbcrse\_subj\_code as subj\_code,  count(a.\*),  MAX(a.scbcrse\_eff\_term)  FROM scbcrse as a  group by scbcrse\_subj\_code  order by 2, a.scbcrse\_subj\_code | {scbcrse=  {scbcrse\_subj\_code=[@...],  scbcrse\_eff\_term=[@...]}  } | In these two queries, the input table and columns are exactly the same, so the Table Dictionary for these two queries are also exactly the same. |
| Queries Inside of Queries | | |
| SELECT first FROM third  union select second from fifth  union select fourth from sixth  union select seventh from eighth | {sixth={fourth=[@11,70:75='fourth',<205>,1:70]},  third={first=[@1,8:12='first',<77>,1:8]},  eighth={seventh=[@16,102:108='seventh',<205>,1:102]},  fifth={second=[@6,39:43='second',<205>,1:39]}  } |  |
| SELECT first FROM third  intersect select second from fifth  intersect select fourth from sixth  intersect select seventh from eighth | {sixth={fourth=[@11,70:75='fourth',<205>,1:70]},  third={first=[@1,8:12='first',<77>,1:8]},  eighth={seventh=[@16,102:108='seventh',<205>,1:102]},  fifth={second=[@6,39:43='second',<205>,1:39]}  } | Structurally, this query and the one previous will produce exactly the same Symbol Table and Table Dictionary. This is because the Union and Intersect operations affect these structures in exactly the same way. The only difference is the outermost reference label which indicates how the symbols are combined. |
| SELECT aa.scbcrse\_coll\_code FROM scbcrse aa  WHERE aa.scbcrse\_subj\_code = courses.subj\_code  AND aa.scbcrse\_crse\_numb = courses.crse\_numb  AND aa.scbcrse\_eff\_term = (SELECT MAX(scbcrse\_eff\_term)  FROM scbcrse  WHERE scbcrse\_subj\_code = courses.subj\_code  AND scbcrse\_crse\_numb = courses.crse\_numb  AND scbcrse\_eff\_term <= courses.term) | {  courses={  term=[@51,324:330='courses',<205>,1:324],  crse\_numb=[@20,120:126='courses',<205>,1:120],  subj\_code=[@12,74:80='courses',<205>,1:74]},  scbcrse={  scbcrse\_coll\_code=[@1,7:8='aa',<205>,1:7],  scbcrse\_crse\_numb=[@16,97:98='aa',<205>,1:97],  scbcrse\_eff\_term=[@24,143:144='aa',<205>,1:143],  scbcrse\_subj\_code=[@8,51:52='aa',<205>,1:51]}  } | An example of a subquery embedded as a predicand in a Where condition. The definition of the subquery’s Symbol Table appears at the same level as other tables in the main query. The fact that the query is part of a condition statement is not apparent in the Symbol Table. When interpreting this table, actual tables refer to their columns directly, while the columns available from the subquery must be retrieved from their “interface” subtrees. |
| SELECT first\_item,  (SELECT item  FROM sgbstdn  WHERE sgbstdn\_levl\_code = 'US') AS INTERNATIONAL\_IND  FROM sgbstdn | {  sgbstdn={  sgbstdn\_levl\_code=[...],  item=[…],  first\_item=[…]}  } | The query shows how an embedded query can act as a predicand. The table dictionary correctly locates the inner-most table references and constructs the unified table dictionary even though some columns appear in different levels of the combined query. |
| SELECT 🡨 **query1**  b.att1,  b.att2  from  (SELECT 🡨 **query0**  a.col1 as att1,  a.col2 as att2  FROM tab1 as a  WHERE a.col1 <> a.col3) AS b | {tab1={  col2=[@17,53:53='a',<205>,1:53],  col3=[@31,100:100='a',<205>,1:100],  col1=[@27,90:90='a',<205>,1:90]}} | When the embedded subquery is used in the From or Join statements, then it must be treated differently than when it is used in other contexts (as in the prior examples, as a predicand). This is so that the data type can disambiguate between the role the query plays in the context of the query it is embedded in (query0), and its own internal construction and symbol table (def\_query0). Within the context of query1, query0 acts as just another table or relation, with a set of contributing columns referenced and handled in the normal way. However, in its own right, it has an embedded Symbol Table which defines it. Note, however, that the Table Dictionary correctly understands that the query0 does not represent an external relationship, hence it only contains the references from table tab1. |
| Insert Queries | | |
| insert into  employees  (emp\_sales\_count,  redder)  values  (select  acct\_sales\_count + 1,  greener  FROM accounts) | {accounts={  acct\_sales\_count=[…],  greener=[...]},  employees={  emp\_sales\_count=[…],  redder=[…]}} |  |
| Update Queries | | |
| update  this\_table  set outputA = column1,  outputB = column2,  outputC = column3  from that\_table  where  this\_table.key=that\_table.key | {  that\_table={  key=[…]},  this\_table={  outputC=[...],  outputA=[@3,22:28='outputA',<205>,1:22],  outputB=[@7,41:47='outputB',<205>,1:41],  key=[@17,101:110='this\_table',<205>,1:101]}  } | <? The column1, column2, column3 columns are missing because the unknowns were not captured. should obviously have been put into the “that\_table” dictionary and symbol table, since that is the only place they could have come from. Need to add “unknown” cleanup logic to the update query step… ?> |
| With Queries | | |
|  |  |  |
|  |  |  |

# Query Interface

The Interface object is an array of strings representing the list of output columns of the SQL statement. This is relatively straight forward to describe as the set of columns produced by the execution of the query (or insert or update statement).

For select statements, this is the set of columns (specifically the alias names, if such are defined), of the selection list. Select statements can be quite complex, with many embedded subqueries combined in various ways. But the Interface it produces is simply the outermost columns. Because this is an aliased name, the names of the interface may not correspond to any pre-existing column on any existing table.

For update statements, this is the set of columns of the target table being set. For insert statements, this is the set of columns of the target table in the insert list (similar to the select statement).

The difference between the interface of the select statement and of the update and insert statements is that for the latter two, the interface corresponds to actual columns in an actual table.

## Select Statement Interface

The following example select statement is highlighted to show the values that make up the SQL statement’s interface.

Select column1 as outputA, column2 as outputB, column3 from this\_table

Given this query, the interface request will return the following array of strings:

[OUTPUTA, OUTPUTB, COLUMN3]

## Insert Statement Interface

<? This does not work today ?>

The following example insert statement is highlighted to show the values that make up the SQL statement’s interface.

Insert into this\_table (column1, column2, column3) values (‘a’, ‘b’, ‘c’)

Given this query, the interface request will return the following array of strings:

[COLUMN1, COLUMN2, COLUMN3]

## Update Statement Interface

<? This does not work today ?>

The following example update statement is highlighted to show the values that make up the SQL statement’s interface.

update this\_table set outputA = column1, outputB = column2, outputC = column3

from that\_table where this\_table.key=that\_table.key

Given this query, the interface request will return the following array of strings:

[OUTPUTA, OUTPUTB, OUTPUTC]

# Advanced Language Features

Advanced:

Condition Over Children Population

Determine the truth of an aggregated condition over a set of instances of children of a parent entity.

Examples:

A) For [Student], at least 3 [Student Coursework] with [Final Grade] = 'F'

A) Count the number of courses a student has failed, and determine if this number exceeds a threshold value.

B) For [Student], any [Student Degree] with [Awarded Date] is not null

B) Determine if the student has ever graduated with an undergraduate degree.

C) Filter [Student], all [Student Course] with [Status] = 'INCOMPLETE'

C) If all of the student's coursework has an incomplete status, filter out the student.

## Traversal and Retrieval (Lookup)

The following language constructs permit the retrieval or creation of values to be included in one part of the domain by traversing to other parts of the domain. These constructs provide a simplified (and incomplete) implementation of relational algebraic concepts, providing short hand for certain very common traversals without the complexity of more complete languages (such as SQL). The table shows comparisons between PUML statements and their possible SQL equivalent statements. The primary difference is that SQL requires explicit references to key column names, while PUML assumes implicit relationships, and allows the configuration to fill in the details about actual equivalences.

Primarily, for PUML, the problem can be described as the need to traverse a chain of relationships between sets of data in order to retrieve values from other parts of the domain. A traversal begins and ends at one part of the domain, and can walk to any other part of the domain, so long as there is a path of relationships. A traversal must follow from one instance of an entity to one instance of the next type of entity. A PUML traversal is, therefore, related to but is not the same as a relational operation such as a join or cross product.

The structure of a specific PUML traversal statement begins with a reference to the value to be retrieved at the farthest step of the traversal, and then proceeds to define each step in the path in turn. Steps are taken by walking across relationships between entities in the domain, while being mindful of (and explicitly defining the rule for choosing amongst) various branches in the path. Some steps, when taken, such as from a child entity to a parent entity, offer only a single choice. However, when the direction of the step is reversed, from a parent to a child entity, some additional language constructs are necessary in order to specify which of the many child instances the traversal should choose.



The figure above depicts a simple education domain, and will act as the model against which the examples are written.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Description | Example Bind to  [Entity].[Attribute] | PUML Formula | SQL Equivalent |
| Type: Self Reference | Find the value of an attribute on the current entity and bind it to another attribute on the same row. Examples: useful if two attributes should have the same value, and the logic is complex. | [College].[College Name] | [College Code] | select college\_cd as college\_name, college\_cd as college\_code from college |
| Type: Self Aggregation | Summarize over the set or a subset of entities to create an attribute value. Examples: find the average credits per term for a student. | [Student Coursework].[Average Credits Per Term] | average([Credits]) by [Student Id] | select a.\*, b.avg\_credits\_per\_term from student\_coursework a join (select student\_id, avg(credits) as avg\_credits\_per\_term from student\_coursework group by student\_id) b on (a.student\_id = b.student\_id) |
| Type: Lookup Parent from Child | Using the set of correct key values from a child entity, lookup the corresponding parent entity and retrieve one of the parent's attribute values. Example: | [Student Degree].[Major Name] | [Major].[Major Name] | select a.\*, b.major\_name from student\_degree a join major\_lkp b on (a.major\_cd=b.major\_id) |
| Type: Lookup Child Aggregate from Parent | Using  a child entity, create an aggregate of some value and add it as a value on the parent entity. Example: Sum all of the credits on courses completed by a student and place the total on the student. | [Student].[Total Credits] | sum([Student Coursework].[Credits] | select a.\*, b.total\_credits from student\_coursework a join (select student\_id, sum(credits) as total\_credits from student\_coursework group by student\_id) b on (a.student\_id=b.student\_id) |
| Type: Select Self With Best Value | From a set of similar records, determine which has/have the best value for some attribute and mark each row as having or not having that value. Example: Select the Student's highest scoring Exam score. | [Student Exam].[Best Exam Score] | ([Score] = max([Score] by [Student Id], [Exam Code]) | select a.\*, b.best\_exam\_score from student\_exam a join (select student\_id, exam\_cd, max(score) as best\_exam\_score from student\_exam group by student\_id, exam\_cd) b on (a.student\_id=b.student\_id and a.exam\_cd=b.exam\_cd and a.score=b.best\_exam\_score) |
| Type: Lookup Ancestor Value Through Chain From Child | From a child entity, lookup through one or more other parent levels to retrieve one of the ancestor's attribute values. Example: | [Student Degree].[College Name] | [College].[Name] via [Major] to [Department] to [College] | select a.\*, d.college\_name from student\_degree a join major\_lkp b on (a.major\_cd=b.major\_cd) join department\_lkp c on (b.department\_cd=c.department\_cd) join college\_lkp d on (c.college\_cd=d.college\_cd) |
| Type: Traverse Across Selected Child | Walk from one parent to another parent by selecting the child that matches a specific condition. The condition should be specific enough to define a single child, to obtain deterministic result. Example: Obtain the major name for the student's most current, primary degree. | [Student].[Current Major Name] | [Major].[Major Name] via [Student Degree] with ([Term Id] = max([Term Id] and [Primary Degree] = 'Y') | select a.\*, c.major\_name from student a join (select student\_id,  major\_cd from student\_degree x join (select student\_id, max(term\_id) as max\_term\_id from student\_degree where primary\_degree='Y' group by student\_id) y on (x.student\_id=y.student\_id and x.term\_id=y.max\_term\_id  where primary\_degree='Y') b on (a.student\_id=b.student\_id) join major c on (b.major\_cd=c.major\_cd) |
| Type: Traverse Across Children and Parents | Walk from one parent to an ancestor by selecting the child that matches a specific condition and then walking to the ancestor. Example: Obtain the college name for the student's most current, primary degree. | [Student].[Current College Name] | [College].[College Name] via [Student Degree] with ([Term Id] = max([Term Id] and [Primary Degree] = 'Y') to [Major] to [Department] to [College] | select a.\*, e.college\_name from student a join (select student\_id,  major\_cd from student\_degree x join (select student\_id, max(term\_id) as max\_term\_id from student\_degree where primary\_degree='Y' group by student\_id) y on (x.student\_id=y.student\_id and x.term\_id=y.max\_term\_id where primary\_degree='Y') b on (a.student\_id=b.student\_id) join major c on (b.major\_cd=c.major\_cd) join department\_lkp d on (c.department\_cd=d.department\_cd) join college\_lkp e on (d.college\_cd=e.college\_cd) |
| Type: Lookup Parent from Child Using Attributes | Using a set of explicitly named Attributes from the child, lookup into the parent and retrieve a specific value | [Student Coursework].[Standard Grade] | Current [Standard Grade].[Standard Grade Code] using .[Grade Code] | select a.\*, std.std\_grade\_code from coursework a join (select std\_grade\_code, grade\_code from standard\_grade) std on (std.grade\_code = a.grade\_code) |