PartiQL Specification

The PartiQL Specification Committee

August 1, 2019

Contents

PartiQL Specification License

This Agreement sets forth the terms under which Amazon is making the PartiQL Specification available to you.

Copyrights Permission is hereby granted, free of charge, to you to use, copy, publish, and/or distribute, the Specification subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies of the Specification.

You may not modify, merge, sublicense, and/or sell copies of the Specification.

Patents Subject to the terms and conditions of this Agreement, Amazon hereby grants to you a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer implementations of the Specification that implement and are compliant with all relevant portions of the Specification (“Compliant Implementations”). Notwithstanding the foregoing, no patent license is granted to any technologies that may be necessary to make or use any product or portion thereof that complies with the Specification but are not themselves expressly set forth in the Specification.

If you institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that Compliant Implementations of the Specification constitute direct or contributory patent infringement, then any patent licenses granted to You under this Agreement shall terminate as of the date such litigation is filed.

THE SPECIFICATION IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL AMAZON BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SPECIFICATION, IMPLEMENTATIONS OF THE SPECIFICATION, OR THE USE OR OTHER DEALINGS IN THE SPECIFICATION.

Any sample code included in the Specification, unless otherwise specified, is licensed under the Apache License, Version 2.0.

1 Introduction

Draft Status This document is currently a working draft and subject to change. Certain sections are marked as "work in progress" (WIP) and will be expanded soon.

Audience This document presents the formal syntax and semantics of PartiQL. It is oriented to PartiQL query processor builders who need the full and formal detail on PartiQL.

SQL users who are not interested in the full detail and the complete formalism but are interested in learning how PartiQL extends SQL may also read the tutorial. Unlike this formal specification, the tutorial has a "how to" orientation and is primarily driven by examples.

PartiQL core and PartiQL syntactic sugar In the interest of precision and succinctness, we tier the PartiQL specification in two layers: The PartiQL core is a functional programming language with composable aspects. Three aspects of the PartiQL core syntax and semantics are characteristic of its functional orientation: Every (sub)query and every (sub) expression input and output PartiQL data. Second, each clause of a SELECT query is itself a function. Third, every (sub)query evaluates within the environment created by the database names and the variables of the enclosing queries.

Then we layer "syntactic sugar" features over the core. Commonly, syntactic sugar achieves well-known SQL syntax and semantics. Formally, every syntactic sugar feature is explained by reduction to the core.

2 Data Model

1	*value*	→	*absent_value*
2		|	*scalar_value*
3		|	*tuple_value*
4		|	*collection_value*
5	*absent_value*	→	`NULL`
6		|	`MISSING`
7	*scalar_value*	→	`` ` `` ***ion_literal*** `` ` ``
8		|	***sql_literal***
9	*tuple_value*	→	`{ }`
10		|	`{` ***string_value*** `:` *value* (`,` ***string_value*** `:` *value*)* `}`
11			*value* cannot be `MISSING`
12	*collection_value*	→	*array_value*
13		|	*bag_value*
14	*array_value*	→	`[` (*value* (`,` *value*)*)? `]`
15	*bag_value*	→	`<<` (*value* (`,` *value*)*)? `>>`

Figure 1: BNF Grammar for PartiQL Values

Figure 1 shows the BNF grammar for PartiQL values. A PartiQL database generally contains one or more PartiQL *named values*. A *name*, is an identifier, such as a table name, that is associated with a PartiQL value. Section 3 defines what these names are, and how SQL qualified names work, in detail.

The type of a value is *absent*, *scalar*, *tuple* or *collection*. Further subtyping applies to scalars, tuples, and collections. We will often use the name *complex value* to refer to any non-scalar and non-absent value. That is, complex values include *tuples* and *collections*. A tuple is a set of attribute name/value pairs, where each name is a string (as in SQL). A tuple in the data model is unordered. A conventional SQL tuple is an ordered tuple since the schema dictates the order of the attributes and certain SQL operations may use the order—support for this is described in detail in Section 15.

PartiQL's data model extends SQL to Ion's type system to cover schema-less and nested data. Such values can be directly quoted with `quotes.

Unlike SQL, PartiQL allows the possibility of duplicate attribute names, in the interest of compatibility with non-strict JSON/Ion datasets. However PartiQL does not encourage duplicate attribute names; navigation into tuples with the conventional dot notation (Section 4) is tuned to the assumption that the attribute names are unique.

A *collection_value* is either an ordered or unordered (BNF lines 12–13). Both arrays and bags may contain duplicate elements. An array is ordered (similar to a JSON array or Ion list) and each element is accessible by its ordinal position. (See specifics of access by position in Section 4.) Arrays are delimited with `[ ]`. For example, the value of the attribute `configurationItems` in Figure 2 is an array. Arrays have size, which is not explicitly denoted but is implied by the number of elements in the array. For example, the size of the `configurationItems` in Figure 2 is 2. The first element of an array corresponds to index 0; the last element corresponds to index size minus one.

In contrast, a bag is unordered (similar to a SQL table) and its elements cannot be accessed by ordinal position. Bags are denoted with `<<` and `>>`.

Finally, note that PartiQL has two kinds of absent values: `NULL` and `MISSING`. The motivation is as follows: Unlike SQL, where a query that refers to a non-existent attribute name is expected to fail during compilation, in semi-structured data one expects a query to operate even if some of the tuples do not define some of the attributes that the query's paths mention. Hence PartiQL contains the special value `MISSING` (BNF line 6), which is the path result in cases where navigation fail to bind to any information. The distinction between `MISSING` and `NULL` enables retaining the original distinction between a missing attribute and a null-valued attribute. The utility of `MISSING` (as opposed to just having `NULL`) will become further apparent when navigation into semi-structured data and construction of semi-structured results is discussed.

The value `MISSING` may not appear as an attribute value. Notice that in the interest of readability, the syntax of Figure 1 does exclude these cases; rather the "*value* cannot be `MISSING`" restrictions (BNF line 11) indicate that `MISSING` cannot appear as an attribute value.

Comparisons to the relational model In summary, the PartiQL data model extended the SQL data model in the following ways:

1. The elements of an array/bag can be any kind of value—not just tuples. Furthermore they can be heterogeneous. That is, there are no restrictions between the elements of an array/bag. For example, the two tuples in `configurationItems` array of are *heterogeneous* because: (i) each tuple has a different set of attributes (for example, the second tuple has `configurationStateId` while the first does not), (ii) an attribute of a first tuple may map to some type while the same attribute in a second tuple may map to another type.
2. More broadly, unlike SQL where the values are tables that have homogeneous tuples that have scalars, PartiQL complex values are *arbitrary compositions of arrays, bags and tuples*. E.g., the top level value of Figure 2 is a tuple and the `configurationItems` array has two heterogeneous tuples. Note that in this example, the top-level name refers to a value that is *not* a bag (e.g. a table).
3. There is a distinction between null-valued attributes and missing attributes.
4. PartiQL makes an explicit distinction between arrays and bags, where the former have order and their elements can be addressed by ordinal position. [1]

[1] Despite the "SQL table is a bag" and "the results of an SQL query is a table" statements of SQL textbooks, SQL also recognizes that the result of a query that has an `ORDER BY` is a list, i.e., an ordered collection of tuples.

```
{
  'fileVersion':'1.0',
  'configurationItems':[
    {
        'configurationItemCaptureTime': `2016-08-03T08:56:52.415Z`,
        'resourceId':'h-0337bfe6793cf9e0c',
        'configuration':{
          'hostId':'h-0337bfe6793cf9e0c',
          'hostProperties':{
              'sockets':2,
              'cores':20,
              'totalVCpus':32,
              'instanceType':'m4.medium'
          },
        'tags':{
          'CostCenter':'Prod'
        },
    },
    {
        'configurationItemCaptureTime':`2016-08-03T09:41:56.906Z`,
        'resourceId':'h-0337bfe6793cf9e0c',
        'configurationStateId':3,
        'configuration':{
          'hostId':'h-0337bfe6793cf9e0c',
          'autoPlacement':'off',
          'hostProperties':{
              'sockets':2,
              'cores':20,
              'totalVCpus':32,
              'instanceType':'m3.medium'
          },
        'tags':{
        },
    }
  ]
}
```

Figure 2: An Example of a PartiQL Value

3 Queries, Environments and Binding Tuples

```
1  query
2    → sfw_query
3    | expr_query
4  sfw_query
5    → (WITH query AS variable)?
6      select_clause
7      from_clause
8      (WHERE expr_query)?
9      (GROUP BY expr_query (AS variable)?
10       (, expr_query (AS variable)?)*)?
11       GROUP AS variable
12     (HAVING expr_query)?
13     ((OUTER)? (UNION|INTERSECT|EXCEPT) ALL? sfw_query)?
14     ((ORDER BY (expr_query (ASC|DESC)? order_spec?
15           (, expr_query (ASC|DESC)? order_spec?)*)? )
16       | PRESERVE)?
17     (LIMIT expr_query)?
18     (OFFSET expr_query)?
19 expr_query
20   → ( sfw_query )
21   | path_expr
22   | function_name ( (expr_query (, expr_query)*)? )
23   | { (expr_query : expr_query (, expr_query : expr_query)*)? }
24   | [ (expr_query (, expr_query)*)? ]
25   | << (expr_query (, expr_query)*)? >>
26   | sql_scalar_expr
27   | value_constant
28 path_expr
29   → variable
30   | ( expr_query )
31   | path_expr . attr_name
32   | path_expr [ expr_query ]
33   | path_expr . *
34   | path_expr [ * ]
```

Figure 3: BNF Grammar for PartiQL Queries

PartiQL may be seen as a functional programming language with composable semantics. Three aspects of the PartiQL syntax and semantics are characteristic of its functional orientation: First, every (sub-)query and every (sub-)expression input and output PartiQL data. Second, each clause of an SFW query (**SELECT-FROM-WHERE**) is itself a function. Third, every (sub-)query evaluates within an *environment* created by the database names and the variables of the enclosing queries.

3.1 Basics of PartiQL Syntax

A PartiQL query is either an *SFW query* (i.e. **SELECT-FROM-WHERE**-..., (Figure 3, lines 4–18) of the grammar of Figure 3) or an *expression query* (also called *simple expression* in the rest, lines 19–34) such as a path expression (Figure 3, lines 28–34) or a function invocation. Unlike SQL expressions, which are restricted to outputting scalar and null values, PartiQL expressions output arbitrary PartiQL values, and are fully composable within larger SFW queries and expressions. Indeed, PartiQL allows the top-level query to also be an expression query, not just a SFW query as in SQL.

An PartiQL (sub)query is evaluated within an environment, which provides variable bindings (as defined next).

3.2 Environments

```
1  bind_name       →  global_name
2                  |  variable
3  qualified_name  →  identifier (. identifier)+
4  variable_name   →  identifier
5  identifier      →  ($ | _ | letter) ($ | _ | letter | digit)*
6                  |  " quoted_identifier_body "
```

Figure 4: BNF Grammar for PartiQL Names

Each PartiQL (sub-)query and PartiQL (sub-)expression q is evaluated within the *database environment* ρ_0 created by the database names and the *variables environment* ρ created by the defined query variables. The pair of these

environments, (ρ_0, ρ) is collectively called the *bindings environment*.

In either case, an environment is a *binding tuple* $\langle x_1 : v_1, \ldots, x_n : v_n \rangle$, where each x_i is a *bind name* (Figure 4, lines 4–18) that is unique and binds to the PartiQL *value* v_i. The two distinct environments may also be thought of as *global* (the database object names) and *local* (the lexically defined variables in a particular scope of the query).

Similarly, for a given q at compile (i.e. planning) time, a *database type environment*, Γ_0, and *variables type environment*, Γ are defined. The type environment is a *binding tuple* $\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle$, where each x_i is a *name* that is unique and binds to the PartiQL *type* τ_i. For schema-less values, τ can be considered the union of any possible type (for which all operations are *potentially* applicable). This is discussed in more detail in Section 15.

Qualified names (Figure 4, line 3) only ever appear in the *database environment*. Lexically defined *variable names* (Figure 4, line 4) are always just simple identifiers (Figure 4, lines 5–6) . For example, a relational database might define a compound name `mydb.log`, where `mydb` is the schema (and not actually a value) and `log` could be a table name within that schema. Note, that a qualified name is distinct from a quoted identifier that contains a dot. Thus, the qualified name `mydb.log` is distinct from the bind name `"mydb.log"`.

Example 1. Let us assume that we evaluate the following query on the database of Figure 2, whose top-level value is named `mydb.log`.

```
SELECT x.resourceId
FROM mydb.log.configurationItems x
```

The query is evaluated within the database environment

$$\rho_0 = \langle \texttt{mydb.log} : \texttt{\{ 'fileversion':'1.0', 'configurationItems':...\}} \rangle$$

and the variables environment $\rho_1 = \langle\rangle$. Notice the database environment ρ_0 has a single name/value pair, which corresponds to the only name (`mydb.log`) of the database of Figure 2. The variables environment has no name/value pair because the above query is not a subquery of a larger query.

Next, consider the subexpression `x.resourceId` of the example's query. This subexpression will, generally, be evaluated many times - once for each `x`. Technically, each time it is evaluated within the same database environment ρ_0 and within a variables environment $\rho_2 = \langle \texttt{x} : \ldots \rangle$, i.e., a variables environment that defines the variable `x`.

Remark on relationship of binding tuples to PartiQL tuples A binding tuple is similar to a PartiQL tuple, if you think of the bind names as attribute names. The characterization "binding" pertains to its use in the semantics (e.g. an association of names to types) and the fact that qualified names are not reified in the PartiQL data model, and we have a representation. As we will see collections of binding tuples will be homogenous, i.e., they will all have the same "attribute" names. Also important, is that when we represent binding tuples we explicitly represent a variable with a `MISSING` value, as opposed to omitting it because the lack of a variable name is distinct from a variable whose value is `MISSING`. For example, we write $\langle \texttt{x} : \texttt{1}, \texttt{y} : \texttt{MISSING} \rangle$, instead of $\langle \texttt{x} : \texttt{1} \rangle$.•

Evaluation in environment The notation $(\rho_0, \rho) \vdash q \rightarrow v$ denotes that the PartiQL query q evaluates to the value v when evaluated within the database environment ρ_0 and the variables environment ρ, i.e. when every variable of q is instantiated by its binding in ρ and each database name is instantiated to its value in ρ_0. For example, consider the query `x + y / 2`, the database environment $\rho_0 = \langle \texttt{x} : \texttt{5} \rangle$ and the variables environment $\rho = \langle \texttt{y} : \texttt{3} \rangle$. Then $\rho_0, \rho \vdash \texttt{(x + y)/2} \rightarrow \texttt{5+3/2} \rightarrow \texttt{4}$.

3.3 The semantics of each clause of an SFW query explained as input and output of binding tuples

The semantics of PartiQL are shorter than the semantics of SQL itself—despite being backwards compatible with SQL. The key reason is that the semantics of each clause of an SFW query in the PartiQL core can be understood in isolation from the other clauses. A clause is simply a function that inputs and outputs binding tuples. Thus the specifics of how the binding tuples of a query and of its subqueries are produced are a central part of the semantics. At a high level (which will be elaborated upon later) the construction of binding environments proceeds as follows:

1. When a query is submitted to a database, it is evaluated in an empty variables environment $\rho = \langle\rangle$.
2. The `FROM` clause of a SFW query produces new environments by concatenating bindings of the `FROM` variables to the environment of its SFW query, as explained below.
   The subqueries that appear in the `WHERE`, `SELECT`, etc clauses are evaluated in these new environments. The optional `GROUP BY` clause also produces additional variable bindings, as explained in Section 11.

$\rho_0 : \langle$ `mydb.r : [3, 'x' ], mydb.s : ≪{'a':1, 'b':2}, {'a':3} ≫` $\rangle$
$\rho : \langle\rangle$

```
FROM mydb.r AS x, mydb.s AS y
```
$B^{out}_{\texttt{FROM}} = B^{in}_{\texttt{WHERE}} =$
```
≪⟨x:3, y:{'a':1, 'b':2}⟩
 ⟨x:3, y:{'a':3}⟩
 ⟨x:'x', y:{'a':1, 'b':2}⟩
 ⟨x:'x', y:{'a':3}⟩
≫
```
```
WHERE x > y.b
```
$B^{out}_{\texttt{WHERE}} = B^{in}_{\texttt{SELECT}} =$ `≪ ⟨x:3, y:{'a':1, 'b':2}⟩ ≫`
```
SELECT x AS foo, y.a AS bar
```
Result: `≪ {foo:3, bar:1} ≫`

Figure 5: An Example SFW Query with Flow of Binding Tuples

SFW query clauses as operators that input/output binding tuples Similar to SQL semantics, the clauses of an PartiQL SFW query are evaluated in the following order: `WITH`, `FROM`, `LET`, `WHERE`, `GROUP BY`, `HAVING`, `LETTING` (which is special to PartiQL), `ORDER BY`, `LIMIT / OFFSET` and `SELECT` (or `SELECT VALUE` or `PIVOT`, which are both special to ion PartiQL). [2]

Using the example of Figure 5, we illustrate how the clauses of an SFW query input and output binding tuples. In the Figure 5, the `FROM`, `WHERE` and `SELECT` clauses of the example query are displayed apart from each other so that the example can also illustrate the binding tuples that flow from the one clause to the next.

The query is evaluated within the bindings environment (ρ_0, ρ) shown at the top of Figure 5. Consequently, the `FROM` clause is evaluated in the same environment. Thereafter the `FROM` clause outputs the bag of binding tuples $B^{out}_{\texttt{FROM}}$, which has four binding tuples in the example. In each binding tuple of $B^{out}_{\texttt{FROM}}$, each variable of the `FROM` clause is bound to a value. There are no restrictions that a variable binds to homogenous values across binding tuples. In the example, `x` binds to two values that are heterogeneous: some bindings of `x` bind to a number, while others to a string. It would also be possible that a variable binds to, say, a scalar in one binding, while the same variable binds to a complex value in another binding.

Each subsequent clause inputs a bag of binding tuples, evaluates the constituent expressions of the clause (which may themselves contain nested SFW queries), and outputs a bag of binding tuples that is in turn input by the next clause. For instance, the `WHERE` clause inputs the bag of binding tuples that have been output by the `FROM` clause ($B^{out}_{\texttt{FROM}} = B^{in}_{\texttt{WHERE}}$), and outputs the subset thereof that satisfies the condition expression of the `WHERE` clause. This subset is the $B^{out}_{\texttt{WHERE}} = B^{in}_{\texttt{SELECT}}$.

In particular, the `WHERE`'s condition is evaluated once for each input binding tuple b in $B^{in}_{\texttt{WHERE}}$. In general, each evaluation is done within the bindings environment $(\rho_0, \rho \| b)$, i.e., the concatenation of the binding tuple ρ (where ρ is the binding environment of the SFW query) with the binding tuple b that has the variables of the `FROM` clause. In the particular example $\rho \| b$ is simply b since $\rho = \langle\rangle$. The condition `x > y.b` is evaluated once for each of the four input binding tuples of $B^{in}_{\texttt{WHERE}}$. The variables environment of the first evaluation is:

$$\rho = \langle \texttt{x} : 3, \texttt{y} : \texttt{\{'a':1, 'b':2\}} \rangle$$

The `WHERE` condition evaluates to `true` for the first binding tuple of $B^{in}_{\texttt{WHERE}}$, since

$$(\rho_0, \rho) \vdash \texttt{x > y.b} \rightarrow \texttt{3 > \{'a':1, 'b':2\}.b} \rightarrow \texttt{true}$$

Thus the first binding tuple of $B^{in}_{\texttt{WHERE}}$ is output from the `WHERE` and is input to `SELECT`.

The pattern of "input bag of binding tuples, evaluate constituent expressions, output bag of binding tuples" has a few exceptions: First, the `ORDER BY` clause inputs a bag of binding tuples and outputs an array of binding tuples. Second, a `LIMIT/OFFSET` clause need not evaluate its constituent expression for each input binding tuple. For example a "`LIMIT 10`" clause that inputs an array with 100 binding tuples need not access binding tuples 11-100.

Finally, the `SELECT` clause is responsible for converting from binding tuples to collections of arbitrary PartiQL elements. The `SELECT` inputs a bag (or array, if `ORDER BY` is present) of binding tuples, and outputs the SFW query's result, which is a bag (resp. array) with exactly one element for each input binding tuple. In the example, the `SELECT` expressions `x` and `y.a` are evaluated once for each of the input binding tuples of $B^{in}_{\texttt{SELECT}}$, which in this example happen to be just one binding tuple.

[2] PartiQL also supports a syntax improvement where `SELECT` is optionally written as the last clause since, anyway, that's the proper way to read an SQL query.

Finally, notice that the above discussion of SFW queries did not capture the set operators `UNION`, `INTERSECT` and `EXCEPT`. As is the case with SQL semantics too, the coordination of `ORDER BY` with the set operators requires attention, as discussed in Section **??**.

PartiQL clauses as operators In summary, each clause of PartiQL is an operator that inputs/outputs binding tuples. As such, we can (and will) present the semantics of each clause separately from the semantics of the other clauses. This is not the case in SQL: Notably, in the presence of aggregation functions the `SELECT`, `HAVING` and `WHERE` cannot be interpreted in isolation; they can only be interpreted along with the `GROUP BY` clause.

3.4 Scoping Rules of Variables

As in any programming language, the PartiQL semantics have to deal with issues of variable scope. For example, how are references to `x` resolved in the following query:

```
SELECT x.a AS a
FROM db1 AS x
WHERE x.b IN (SELECT x.c FROM db2 AS x)
```

Since this is an SQL query and PartiQL is backwards compatible to SQL, it is easy to tell that the `x` in `x.c` resolves to the variable `x` defined by the inner query's `FROM` clause.

Technically, this scoping rule is captured by the following handling of binding tuples. The inner `FROM` clause is evaluated with a variables environment $\rho = \langle \texttt{x} : \ldots \rangle$; its `x` is the one defined by the outer `FROM`. Then the inner `FROM` clause outputs a binding $b = \langle \texttt{x} : \ldots \rangle$; this `x` is defined by thinner `FROM`. Then the `x.c` is evaluated in the concatenation $\rho \| b$ and because `x` appears in both ρ and b, the concatenation keeps only the `x` of its right argument. Essentially by putting b as the right argument of the concatenation, the semantics indicate that the variables of b have precedence over synonymous variables in the left argument (which was the ρ).

Generally, given two binding tuples b and b', their concatenation is a binding tuple, denoted as $b \| b'$, that has the variable bindings of both b and b'. This creates the possibility that both b and b' have the same variable x. In this case, the concatenation $b||b'$ will have the $b'.x$ and its value; it will not have the $b.x$ and its value.

Note, the above does not resolve scoping issues resulting from conflicts between the database environment and the variables environment. We resolve these conflicts by explicit rules.

4 Path Navigation

Tuple path navigation A *tuple path navigation* $t.a$ from the tuple t to its attribute a (Figure 3, lines 32–33) returns the value of the attribute a. (We discuss below the corner case where a tuple has multiple attributes `a`.) t is an expression but a is always an identifier (Figure 4, lines 5–6) . For example:

```
{'a': 1, 'b':2}.a ⇔ {'a': 1, 'b':2}."a" → 1
```

Even if there were a variable `a`, bound to `'b'`, the result of the above expression would still be `1`, because the identifier `a` (or `"a"`) is interpreted as the "look for the attribute named `a`" when it follows the dot in a tuple path navigation. The semantics of tuple path navigation do not depend on whether the tuple is ordered or unordered by schema.

Array navigation An *array navigation* $a[i]$ returns the i-th element *when* it is applied on an array a (Figure 3, line 32) and i is an expression that evaluates into an integer. Both a and i are expressions. For example:

```
[2, 4, 6][1+1] → 6.
```

Tuple navigation with array notation The expression $a[s]$ is a shorthand for the tuple path navigation $a.s$ when the expression s is either (a) a string literal or (b) an expression that is explicitly `CAST` into a string. For example:

```
{'a': 1, 'b': 2}['a'] ⇔ {'a': 1, 'b':2}.'a' → 1
```

Similarly:

```
{'attr': 1, 'b':2}[CAST('at' || 'tr' AS STRING)] → 1
```

If s is not a string literal or an expression that is cast into a string, then $a[s]$ is evaluated as an array path navigation. Notice that in the absence of an explicit cast, the navigation $a[e]$ evaluates as an array navigation, even if e ends up evaluating to a string. For example, let us assume that the variable `v` is bound to `'at'` and the variable `w` is bound to `'tr'`. Still, the expression:

```
{'attr': 1, 'b':2}[v || w]
```

does not evaluate to `1`. It is treated as an array navigation with wrongly typed index and it will return `missing`, for reasons explained below.

Composition of navigations Notice that consecutive tuple/array navigations (e.g. `r.no[1]`) navigate deeply into complex values. Notice further that paths consisting of plain tuple and array path navigations evaluate to a unique value.

Tuple navigation in tuples with duplicate attributes When the tuple `t` has multiple attributes `a`, the tuple path navigation `t.a` will return the first instance of `a`. Note that for tuples whose order is defined by schema, this is well-defined, for unordered tuples, it is implementation defined which attribute is returned in *permissive mode* or an error in *type checking mode*, which is described in Section 4.1.

If one wants to access all instances of `a`, she should use the `UNPIVOT` feature instead (see Section 5.2). For example, the following query returns the list of all `a` values in a tuple `t`.

```
SELECT VALUE v
FROM UNPIVOT t AS v AT attr
WHERE attr = 'a'
```

4.1 Tuple path evaluation on wrongly typed data

In the case of tuple paths, since PartiQL does not assume a schema, the semantics must also specify the return value when:

1. t is not a tuple (i.e., when the expression t does not evaluate into a tuple), or
2. t is a tuple that does not have an a attribute.

Permissive mode PartiQL can operate in a permissive mode or in a conventional type checking mode, where the query fails once typing errors (such as the above mentioned ones) happen. In the permissive mode, typing errors are typically neglected by using the semantics outlined next.

In all of the above cases PartiQL returns the special value `missing`. Recall, the `missing` is different from `null`. The distinction enables PartiQL to be able to distinguish between a tuple (JSON object) that lacked an attribute `a` and a tuple (JSON object) whose `a` attribute was `null`. This distinction, coupled with appropriate features on how result tuples are constructed (see `SELECT` clause in Section 6), enables PartiQL to easily preserve (when needed) the distinction between absent attribute and null-valued attribute.

For example, the expression `'not a tuple'.a` and the expression `{'a':1, 'b':2}.noSuchAttribute` evaluate to `missing`.

The above semantics apply regardless of whether the tuple navigation is accomplished via the dot notation or via the array notation. For example, the expression `{'a':1, 'b':2}['noSuchAttribute']` will also evaluate to `missing`.

Type checking mode In the type checking mode and in the absence of schema, PartiQL will fail when tuple path navigation is applied on wrongly typed data.

4.1.1 Role of schema in type checking

In the presence of schema, PartiQL may return a compile-time error when the query processor can prove that the path expression is guaranteed to *always* produce `MISSING`. The extent of error detection is implementation-specific.

For example, in the presence of schema validation, an PartiQL query processor can throw a compile-time error when given the path expression `{a:1, b:2}.c`. In a more important and common case, an PartiQL implementation can utilize the input data schema to prove that a path expression *always* returns `MISSING` and thus throw a compile-time error. For example, assume that `sometable` is an SQL table whose schema does not include an attribute `c`. Then, an PartiQL implementation may throw a compile-time error when evaluating the query:

```
SELECT t.a, t.c FROM sometable AS t
```

Apparently, such an PartiQL implementation is fully compatible with the behavior of an SQL processor. Generally, if a rigid schema is explicitly present, a tuple path navigation error can be caught during compilation time; this is the case in SQL itself, where referring to a non-existent attribute leads to a compilation error for the query.

Notice that operating with schema validation may not prevent all tuple path navigations from being applied to wrongly typed data. The choice between permissive mode versus type checking mode dictates what happens next in these cases: If permissive, the tuple path navigation evaluates into `MISSING`. If in type checking mode, the query fails.

4.2 Array navigation evaluation on wrongly typed data

In the permissive mode, an array navigation evaluation $a[i]$ will result into `missing` in each of the following cases:

- a does not evaluate into an array, or
- i does not evaluate into a positive integer within the array's bounds.

For example, `[1,2,3][1.0]` evaluates to `missing` since `1.0` is not an integer - even though it is coercible to an integer.

In type checking mode, the query will fail in each one of the cases above.

4.3 Additional Path Syntax

The following additional path functionalities are explained by reduction to the basic tuple navigation and array navigation.

Wildcard steps The expression $e[*]$ reduces to (i.e., is equivalent to):

```
SELECT VALUE v FROM e AS v
```

where v is a *fresh variable*, i.e., a variable that does not already appear in the query. Similarly, when the expression $e.*$ is not a `SELECT` clause item of the form $t.*$, where t is a variable, it reduces to:

```
SELECT VALUE v FROM UNPIVOT e AS v
```

where v is a fresh variable. An expression $t.*$, where t is a variable and the expression appears as a `SELECT` clause item, is interpreted according to the `SELECT` clause `*` semantics (Section 6.3.2).

Example 2. The expression:

```
[1,2,3][*] ⇔ SELECT VALUE v FROM [1, 2, 3] AS v
   → <<1, 2, 3>>
```

The expression:

```
{'a':1, 'b':2}.* ⇔ SELECT VALUE v FROM UNPIVOT {'a':1, 'b':2} AS v
   → <<1, 2>>
```

Whereas the following query:

```
SELECT t.* FROM <<{'a':1, 'b':1}, {'a':2, 'b':2}>> AS t
   → <<{'a':1, 'b':1}, {'a':2, 'b':2}>>
```

does not do the transformation with `UNPIVOT`. If one does not want this behavior, `SELECT VALUE` can be used (Section 6).

Path Expressions with Wildcards PartiQL also provides multi-step path expressions, called *path collection expressions*. Their semantics is a generalization of the semantics of a path expression with a single $[*]$ or $.*$. Consider the path collection expression:

$$ew_1p_1 \ldots w_np_n$$

where e is any expression; $n > 0$; each *wildcard step* w_i is either $[*]$ or $.*$; each *series of plain path steps* p_i is a sequence of zero or more tuple path navigations or array navigations (potentially mixed).

Then the path collection expression is equivalent to the SFW query

```
SELECT VALUE v_n.p_n
FROM
    u_1 e AS v_1,
    u_2 @v_1.p_1 AS v_1,
    ...,
    u_n @v_{n-1}.p_{n-1} AS v_n,
```

where each v_i is a fresh variable and each u_i is `UNPIVOT` if w_i is a $.*$ and it is nothing if w_i is a $[*]$. Intuitively v_i corresponds to the i-th star.

Example 3. According to the above, consider the following query:

```
SELECT VALUE foo FROM e.* AS foo
```

reduces to

```
SELECT VALUE foo FROM (SELECT VALUE v FROM UNPIVOT e AS v) AS foo
```

which is equivalent to

```
SELECT VALUE foo FROM UNPIVOT e AS foo
```

Next, consider the path collection expression:

```
tables.items[*].product.*.nest
```

This expression reduces to

```
SELECT
  VALUE v2.nest
FROM
  tables.items AS v1,
  UNPIVOT @v1.product AS v2
```

5 FROM Clause Semantics

The formal semantics of a **FROM** clause describe the collection of binding tuples $B^{out}_{\texttt{FROM}}$ that is output by the **FROM** clause. The semantics specify three cases and essentially extend the tuple calculus that underlies the SQL semantics.

1. The semantics specify what is the core semantics of a **FROM** clause with a single **FROM** item (Sections 5.1 and 5.2). The term "semantics of the **FROM** item f" is synonymous to the term "semantics of a **FROM** clause with the single item f". In either case, we refer to the specification of the collection of binding tuples $B^{out}_{\texttt{FROM}}$ that results from the evaluation of "**FROM** f".
2. Then the semantics specify how multiple **FROM** items combine, according to the core semantics, using the join and outerjoin operations (Sections 5.3, 5.4 and 5.5).
3. Finally, the semantics specify the syntactic sugar structures that are overlaid over the core semantics. Their primary purpose is SQL compatibility.

5.1 Ranging Over Bags and Arrays

Next we define the semantics of a **FROM** clause that has a single **FROM** item and such item ranges over a bag or array. First consider the **FROM** clause:

FROM a **AS** v **AT** p

Let us call v to be the *element variable* and p to be the *position variable*. In the normal case, a is an array $[e_0, \ldots, e_{n-1}]$. The **FROM** clause outputs a bag of binding tuples. For each e_i, the bag has a binding tuple $\langle v : e_i, p : i \rangle$.

Example 4. Consider the following ρ_0 (database environment):

$\rho_0 = \langle$
```
    someOrderedTable:[
        {'a':0, 'b':0},
        {'a':1, 'b':1}
    ]
```
$\rangle$

then the following **FROM** clause:

```
FROM someOrderedTable AS x AT y
```

outputs the bag of binding tuples:

$B^{out}_{\texttt{FROM}} = \ll$ `⟨ x:{'a':0, 'b':0}, y:0 ⟩`
`⟨ x:{'a':1, 'b':1}, y:1 ⟩`
$\gg$

As in SQL, the keyword **AS** is optional. The same applies to all cases below where **AS** appears. If there is no **AT** clause, then the binding tuples have only the element variable. In particular, consider:

FROM a **AS** v

Normally a is a collection, i.e, an array $[e_0, \ldots, e_{n-1}]$ or a bag $\ll e_0, \ldots, e_{n-1} \gg$. In either case, the **FROM** clause outputs a bag. For each e_i, the bag has a binding tuple $\langle v : e_i \rangle$.

Example 5. Consider again the database of Example 4 and then the **FROM** clause

```
FROM someOrderedTable AS x
```

this **FROM** clause outputs:

$$B^{out}_{\texttt{FROM}} = \ll \langle\ \texttt{x:\{'a':0, 'b':0\}}\ \rangle \langle\ \texttt{x:\{'a':1, 'b':1\}}\ \rangle \gg$$

5.1.1 Mistyping Cases

In the following cases the expression in the **FROM** clause item has the wrong type. Under the type checking option, all of these cases raise an error and the query fails. Under the permissive option, the cases proceed as follows

- **Position variable on bags:** Consider the clause:

  FROM b **AS** v **AT** p

  and assume that b is a bag $\ll e_0, \ldots, e_{n-1} \gg$. The output is a bag with binding tuples $\langle v : e_i, p : \texttt{MISSING} \rangle$. The value **MISSING** for the variable p indicates that the order of elements in the bag was meaningless.

- **Iteration over a scalar value:** Consider the query:

  FROM s **AS** v **AT** p

  or the query:

  FROM s **AS** v

  where s is a scalar value. Then s coerces into the bag $\ll s \gg$, i.e., the bag that has a single element, the s. The rest of the semantics is identical to what happens when the lhs of the **FROM** item is a bag.

  Example 6. Consider again the database of Example 4 and the **FROM** clause:

  ```
  FROM someOrderedTable[0].a AS x
  ```

  The expression **someOrderedTable[0].a** evaluates to **0** and, consequently, the **FROM** clause outputs a single binding tuple:

  $$B^{out}_{\texttt{FROM}} = \ll \langle\ \texttt{x:0}\ \rangle \gg$$

- **Iteration over a tuple value:** Consider the query:

  FROM t **AS** v **AT** p

  or the query:

  FROM t **AS** v

  where t is a tuple. Then t coerces into the bag $\ll t \gg$

- **Iteration over an absent value:** Consider the query

```
FROM a AS v AT p
```

or the query

```
FROM a AS v
```

whereas a evaluates into an *absent_value* (i.e., either `MISSING` or `NULL`). In either case the *absent_value* a coerces into the bag $\ll a \gg$. Then the semantics follow the normal case.

Example 7. Consider again the database of Example 4 and the `FROM` clause

```
FROM someOrderedTable[0].c AS x
```

The expression `someOrderedTable[0].c` evaluates to `MISSING` and, consequently, the `FROM` clause outputs the binding tuple:

$B^{out}_{\texttt{FROM}} = \ll \langle$ `x:MISSING` $\rangle \gg$

5.2 Ranging over Attribute-Value Pairs

The `UNPIVOT` clause enables ranging over the attribute-value pairs of a tuple. The `FROM` clause

```
FROM UNPIVOT t AS v AT a
```

normally expects t to be a tuple, with attribute/value pairs $a1 : v1, \ldots, a_n : v_n$. It does not matter whether the tuple is ordered or unordered. The `FROM` clause outputs the collection of binding tuples

$$B^{out}_{\texttt{FROM}} = \ll \langle v : v_1, a : a_1 \rangle \ldots \langle v : v_n, a : a_n \rangle \gg$$

Example 8. Consider the ρ_0:

```
ρ0 = ⟨
    justATuple: {'amzn': 840.05, 'tdc': 31.06}
⟩
```

The `FROM` clause:

```
FROM UNPIVOT justATuple AS price AT symbol
```

outputs:

```
B^out_FROM = ≪⟨price: 840.05, symbol:'amzn'⟩
             ⟨price: 31.06, symbol:'tdc'⟩
             ≫
```

5.2.1 Mistyping Cases

In the following cases the expression in the `FROM UNPIVOT` clause item has the "wrong" type, i.e., it is not a tuple. Under the type checking option, all of these cases raise an error and the query fails. Under the permissive option, the cases proceed as follows:

```
FROM UNPIVOT x AS v AT n
```

whereas x is not a tuple and is not `MISSING`, is equivalent to:

```
FROM UNPIVOT {'_1': x} AS v AT n
```

Effectively, a tuple is generated for the non-tuple value. When x is `MISSING` then the above is equivalent to:

```
FROM UNPIVOT {} AS v AT n
```

remember that a tuple cannot contain `MISSING`. So the present case is equivalent to the empty tuple case.

5.3 Combining Multiple FROM Items with Comma, CROSS JOIN, or JOIN

The `FROM` clause expressions:

l , r $\Leftrightarrow$
l `CROSS JOIN` r $\Leftrightarrow$
l `JOIN` r `ON TRUE` $\Leftrightarrow$

have the same semantics. They combine the bag of bindings produced from the `FROM` item l with the bag of binding tuples produced by the `FROM` item r, whereas the expression r may utilize variables defined by l. Again, the term "the semantics of l `CROSS JOIN` r" is equivalent to the term "the semantics of `FROM` l `CROSS JOIN` r". In both cases, the semantics specify a bag of binding tuples.

Associativity of `CROSS JOIN` We explain the `CROSS JOIN` and "," as if they are left associative binary operators, despite the fact that one can write more than two `FROM` items without specifying grouping with parenthesis. Since the "," and `CROSS JOIN` operators are associative, we may write (as is common in SQL):

f_1, f_2, f_3 $\Leftrightarrow$
f_1 `CROSS JOIN` f_2 `CROSS JOIN` f_3 $\Leftrightarrow$
f_1 `LEFT JOIN` f_2 `ON TRUE CROSS JOIN` f_3 `ON TRUE` $\Leftrightarrow$
(f_1, f_2), f_3 $\Leftrightarrow$
(f_1 `CROSS JOIN` f_2) `CROSS JOIN` f_3 $\Leftrightarrow$
(f_1 `LEFT JOIN` f_2 `ON TRUE`) `CROSS JOIN` f_3 `ON TRUE` $\Leftrightarrow$

Semantics Consider the following:

l `CROSS JOIN r`

unlike SQL, the rhs r of the expression may use variables defined by the lhs item l. The result of this expression for a database environment ρ_0 and variables environment ρ is the bag of binding tuples produced by the following pseudo-code. The pseudo-code uses the function $\texttt{eval}(\rho_0, \rho, e)$ that evaluates the expression e within the environments ρ_0 and ρ, i.e., $\rho_0, \rho \vdash e \rightarrow \texttt{eval}(\rho_0, \rho, e)$.

for each binding tuple b^l in $\texttt{eval}(\rho_0, \rho, l)$
  for each binding b^r in $\texttt{eval}(\rho_0, (\rho \| b^l), r)$
    add $b^l \| b^r$ to the output bag

In other words, the "l `CROSS JOIN` r" outputs all binding tuples $b = b^l \| b^r$, where $b^l \in \texttt{eval}(\rho_0, \rho, l)$ and $b^r \in \texttt{eval}(\rho_0, (\rho \| b^l), r)$. The key extension to SQL is that r is evaluated in the variables environment $\rho \| b^l$, i.e., it can use the variables that were defined by l. The details of the variable scoping aspects are described in Section 3.4.

Example 9. This example simply reminds the tuple calculus explanation of the `FROM` SQL semantics. It does not yet endeavor into special aspects of PartiQL. Consider the following database, which is conventional SQL:

```
ρ0 = ⟨
    customers: [
        {'id': 5, 'name': 'Joe'},
        {'id': 7, 'name': 'Mary'}
    ],
    orders: [
        {'custId': 7, 'productId': 101},
        {'custId': 7, 'productId': 523}
    ]
⟩
```

Then consider the following **FROM** clause, which could be coming from a conventional SQL query:

```
FROM customers AS c, orders AS o
```

Note that in PartiQL this could also be written using the **CROSS JOIN** keyword, and presumably, one would put the sensible equality condition **c.id=o.custId** in the **WHERE** clause. At any rate, this **FROM** clause outputs the bag of binding tuples:

```
B^out_FROM = ≪⟨ c: {'id': 5, 'name': 'Joe'}, o: {'custId': 7, 'productId': 101} ⟩
             ⟨ c: {'id': 5, 'name': 'Joe'}, o: {'custId': 7, 'productId': 523} ⟩
             ⟨ c: {'id': 7, 'name': 'Mary'}, o: {'custId': 7, 'productId': 101} ⟩
             ⟨ c: {'id': 7, 'name': 'Mary'}, o: {'custId': 7, 'productId': 523} ⟩
             ≫
```

Due to scoping rules that will be justified and elaborated in Section 10, when the rhs of a **CROSS JOIN** is a path or a function that uses a variable named n, such variable must be referred as **@**n.

Example 10. Consider the database:

```
ρ0 = ⟨
    sensors: [
        {'readings': [{'v': 1.3}, {'v': 2}]},
        {'readings': [{'v': 0.7}, {'v': 0.8}, {'v': 0.9}]}
    ]
⟩
```

Intuitively, the following **FROM** clause unnests the tuples that are nested within the **readings**.

```
FROM sensors AS s, s.readings AS r
```

```
B^out_FROM = ≪⟨ s: {'readings': [{'v': 1.3}, {'v': 2}]}, r: {v:1.3} ⟩,
             ⟨ s: {'readings': [{'v': 1.3}, {'v': 2}]}, r: {v:2} ⟩,
             ⟨ s: {'readings': [{'v': 0.7}, {'v': 0.8}, {'v': 0.9}]}, r: {'v':0.7} ⟩,
             ⟨ s: {'readings': [{'v': 0.7}, {'v': 0.8}, {'v': 0.9}]}, r: {'v':0.8} ⟩,
             ⟨ s: {'readings': [{'v': 0.7}, {'v': 0.8}, {'v': 0.9}]}, r: {'v':0.9} ⟩,
             ≫
```

5.4 Combining Multiple FROM Items with LEFT JOIN

The **FROM** clause expression:

```
l LEFT CROSS JOIN r ⇔
l LEFT JOIN r ON TRUE
```

replicates SQL's **LEFT JOIN** functionality and, in addition, it also works for the case where the lhs of r uses variables defined from l.

Let's assume that the variables defined by r are $v_1^r, \ldots, v_n^r$. The result of evaluating l **LEFT CROSS JOIN** r in environments ρ_0 and ρ is the bag of binding tuples produced by the following pseudocode, which also uses the **eval** function (See Section 5.3).

```
for each binding b^l in eval(ρ0, ρ, l)
    B^r ← eval(ρ0, (ρ‖b^l), r)
    if B^r is the empty bag
      add b^l‖⟨v_1^r : NULL ... v_n^r : NULL⟩ to the output bag
    else
      for each binding b^r in B^r
          add b^l‖b^r to the output bag
```

Example 11. Consider the database:

```
ρ0 = ⟨
    sensors: [
        {'readings': [{'v':1.3}, {'v':2}]}
        {'readings': [{'v':0.7}, {'v':0.8}, {'v':0.9}]},
        {'readings': []}
      ]
⟩
```

Notice that the value of the last tuple's **readings** attribute is the empty array. The following **FROM** clause unnests the tuples that are nested within the **readings** but will also keep around the tuple with the empty **readings**. (See the last binding tuple.)

```
FROM sensors AS s LEFT CROSS JOIN s.readings AS r
```

```
B^out_FROM = ≪⟨ s: {'readings': [{'v':1.3}, {'v':2}]}, r: {'v':1.3} ⟩,
             ⟨ s: {'readings': [{'v':1.3}, {'v':2}]}, r: {'v':2} ⟩,
             ⟨ s: {'readings': [{'v':0.7}, {'v':0.8}, {'v':0.9}]}, r: {'v':0.7} ⟩,
             ⟨ s: {'readings': [{'v':0.7}, {'v':0.8}, {'v':0.9}]}, r: {'v':0.8} ⟩,
             ⟨ s: {'readings': [{'v':0.7}, {'v':0.8}, {'v':0.9}]}, r: {'v':0.9} ⟩,
             ⟨ s: {'readings': []}, r: NULL ⟩,
             ≫
```

5.5 Combining Multiple FROM Items with FULL JOIN

The **FROM** clause expression:

```
l FULL JOIN r ON *$c$*
```

replicates SQL's **FULL JOIN** functionality. It assumes that (alike SQL) the lhs of r does not use variables defined from l. Thus, we do not discuss further.

5.6 Expanding JOIN and LEFT JOIN with ON

In compliance to SQL, the **JOIN** and **LEFT JOIN** have an optional **ON** clause. The semantics of **ON** can be explained as syntactic sugar over the core PartiQL. They can also be explained by a simple extension of the semantics of Sections 5.3, 5.4, and 5.5. The semantics of:

```
l JOIN r ON c
```

are the following modification of the pseudocode of Section 5.3. The modification is the inclusion of the underlined line.

for each binding b^l in $\texttt{eval}(\rho_0, \rho, l)$
  for each binding b^r in $\texttt{eval}(\rho_0, (\rho \| b^l), r)$
    if $\texttt{eval}(\rho_0, (\rho \| b^l \| b^r), c)$ is true
      add $b^l \| b^r$ to the output bag

The semantics of:

l `LEFT JOIN` r `ON` c

are the following modification of the pseudocode of Section 5.4. In essence, the `LEFT JOIN ON` outputs a tuple padded with `NULL` whenever there is no binding of r that satisfies the condition c.

for each binding b^l in $\texttt{eval}(\rho_0, \rho, l)$
  $B^r \leftarrow \texttt{eval}(\rho_0, (\rho \| b^l), r)$
  $Q^r \leftarrow \ll\gg$
  for each binding b^r in B^r
    if $\texttt{eval}(\rho_0, (\rho \| b^l \| b^r), c)$ is true
      add b^r in Q^r
  if Q^r is the empty bag
    add $b^l \| \langle v_1^r : \texttt{NULL} \ldots v_n^r : \texttt{NULL} \rangle$ to the output bag
  else
    for each binding b^r in Q^r
      add $b^l \| b^r$ to the output bag

5.7 SQL's LATERAL

SQL 2003 used the `LATERAL` keyword to correlate `FROM` clause items. In the interest of compatibility with SQL, PartiQL also allows the use of the keyword `LATERAL`, though it does not do anything more than the comma itself would do. That is "l `, LATERAL` r" is equivalent to "l `,` r".

6 SELECT clauses

Core PartiQL SFW queries have a **SELECT VALUE** clause (in lieu of SQL's **SELECT** clause) that can create outputs that are collections of anything (e.g., collections of tuples, collections of scalars, collections of arrays, collections of mixed type elements, etc.) Section 6.1 describes the **SELECT VALUE** clause.

SQL's well-known **SELECT** clause can be used as a mere syntactic sugar over **SELECT VALUE**, when we consider the top-level query. In particular, Section 6.3 shows that SQL's **SELECT** is the special case where the **SELECT VALUE** produces collections of tuples. Furthermore, when **SELECT** is used as a subquery it is coerced into a scalar or a tuple, in the ways that SQL coerces the results of subqueries.

Section 14 describes **PIVOT**, which can be used instead of **SELECT VALUE**. **PIVOT** creates a tuple, with a data dependent number of attribute/value pairs, where not only the values but the attributes as well could be originating from the data found in the binding tuples.

6.1 SELECT VALUE core clause

The **SELECT VALUE** clause inputs a bag of binding tuples or an array of binding tuples (from the other clauses of the SQL query) and outputs a bag or an array. For example, if the query only has **SELECT VALUE**, **FROM**, and **WHERE** clauses, then the bindings that are output by the **WHERE** clause are input by the **SELECT VALUE** clause. Unlike SQL, the output of a **SELECT VALUE** clause need not be a bag or array of tuples. It is a bag or array of any kind of PartiQL values. For example, it may be a bag of integers, or a bag of arrays, etc. Indeed, the values may be heterogeneous. For example, the output may even be a bag that has both integers and arrays.

The core PartiQL clause:

SELECT VALUE e

inputs a bag or an array (depending on the presence or non-presence of **ORDER BY**) of binding tuples and outputs respectively a bag or an array of values. Let ρ_0 and ρ be the environments of the SFW query. For each input binding tuple $b \in B^{in}_{\text{SELECT}}$, **SELECT VALUE** outputs a value v, where $\rho_0, (\rho \| b) \vdash e \to v$. Notice that PartiQL expressions e (Figure 3 lines 19–34) will typically be tuple or array or bag constructors (Figure 3, lines 23–25), which enable the construction of respective results. In general e can be any expression.

Example 12. This example illustrates a **SELECT VALUE** that creates a collection of numbers.

```
SELECT VALUE 2*x.a
FROM [{'a':1}, {'a':2}, {'a':3}] as x
```

The result is `<<2, 4, 6>>`.

6.1.1 Tuple constructors

A *tuple constructor* is of the form

{a_1:e_1, ..., a_n:e_n}

whereas $a_1 \ldots a_n, e_1 \ldots e_n$ are expressions, potentially being themselves constructors.

Example 13. The query:

```
SELECT VALUE {'a':v.a, 'b':v.b}
FROM [{'a':1, 'b':1}, {'a':2, 'b':2}] AS v
```

results into `<<{'a':1, 'b':1}, {'a':2, 'b':2}>>`.

Treatment of mistyped attribute names It is possible that an expression a_i that computes an attribute name results into a non-string, i.e., a value that is not a legitimate attribute name. In such cases, under the permissive mode the attribute-value pair will be dismissed. Under the type checking mode the query will fail.

Example 14. In the permissive mode, the query:

```
SELECT VALUE {v.a: v.b}
FROM [{'a':'legit', 'b':1}, {'a':400, 'b':2}] AS v
```

results into `<<{'legit':1}, {}>>`. Notice that the attempt to create an attribute named `400` failed, thus leading to a tuple with no attributes.

Treatment of duplicate attribute names It is possible that the constructed tuples contain twice or more the same attribute name.

Example 15. The query:

```
SELECT VALUE {v.a: v.b, v.c: v.d}
FROM [{'a':'same', 'b':1, 'c':'same', 'd':2}] AS v
```

results into `<<{'same':1, 'same':2}>>`. Recall, a `same` path will only pick one of the two values.

6.1.2 Array Constructors

An array constructor has the form:

```
[e0, ..., en−1]
```

where $e_1 \dots e_{n-1}$ are expressions. Notice that the arrays produced by such constructor will always have size n.

Example 16. The query:

```
SELECT VALUE [v.a, v.b]
FROM [{'a':1, 'b':1}, {'a':2, 'b':2}] AS V
```

results into `<<[1, 1], [2, 2]>>`

In the interest of compatibility to SQL, PartiQL also allows array constructors to be denoted with parentheses instead of brackets, when there are at least two elements in the array, i.e., $n \geq 2$:

```
(e0, ..., en−1)
```

See Section 9.2 for uses of this feature in SQL compatibility.

6.1.3 Bag Constructors

A bag constructor has the form:

```
<<e0, ..., en−1>>
```

where $e_1 \dots e_n$ are expressions.

Example 17. The query:

```
SELECT VALUE <<v.a, v.b>>
FROM [{'a':1, 'b':1}, {'a':2, 'b':2}] AS v
```

results into `<< <<1, 1>>, <<2, 2>> >>`.

6.1.4 Treatment of MISSING in SELECT VALUE

`MISSING` may behave differently from `NULL` and differently from scalars. The following itemizes the behavior of `MISSING` in a number of cases:

- **when constructing tuples** Whenever during tuple construction an attribute value evaluates to `MISSING`, then the particular attribute/value is omitted from the constructed tuple.

  Example 18. The query

  ```
  SELECT VALUE {'a':v.a, 'b':v.b}
  FROM [{'a':1, 'b':1}, {'a':2}]
  ```

  results into `<<{'a':1, 'b':1}, {'a':2}>>`.

- **when constructing arrays** Whenever an array element evaluates to `MISSING`, the resulting array will contain a `MISSING`.

  Example 19. The query

  ```
  SELECT VALUE [v.a, v.b]
  FROM [{'a':1, 'b':1}, {'a':2}]
  ```

  results into `<<[1, 1], [2, MISSING]>>`.

  Upon output serialization the `MISSING` will convert to the symbol that the serialization has chosen for serializing `MISSING`.

- **when constructing bags** Whenever an element of a bag evaluates to `MISSING`, the resulting bag will contain a corresponding `MISSING`.

  Example 20. The query

  ```
  SELECT VALUE v.b
  FROM [{'a':1, 'b':1}, {'a':2}]
  ```

  results into `<<1, MISSING>>` because `{'a':2}.b` evaluated to `MISSING`.

  Example 21. The query

  ```
  SELECT VALUE <<v.a, v.b>>
  FROM [{'a':1, 'b':1}, {'a':2}]
  ```

  results into `<< <<1, 1>>, <<2, MISSING>> >>`.

6.2 Pivoting a Collection into a Variable-Width Tuple

The `PIVOT` clause may appear in lieu of `SELECT VALUE`. The `PIVOT` clause outputs a tuple; in contrast, a `SELECT VALUE` outputs a collection (bag or array). The syntax is

```
PIVOT e_v AT e_a c
```

where the other clauses, c, are the usual `FROM`, `WHERE`, etc. The semantics are similar to `SELECT VALUE`. Let ρ_0 and ρ be the environments of the SFW query. For each input binding tuple $b \in B^{in}_{\texttt{PIVOT}}$, `PIVOT` outputs an attribute name/value pair a, v, where the name a is the result of e_a and the value v is the result of e_v. (Technically, $\rho_0, (\rho \| b) \vdash e_a \mapsto a$ and $\rho_0, (\rho \| b) \vdash e_v \mapsto v$.) Regardless of whether $B^{in}_{\texttt{PIVOT}}$ is a bag (i.e., the SFW query did not have an `ORDER BY`) or an array (i.e., the SFW query had an `ORDER BY`), the output tuple is unordered. Schema may be applied extantly to obtain an ordered tuple.

Example 22. The query:

```
PIVOT t.price AT t.symbol
FROM [{'symbol':'tdc', 'price': 31.52}, {'symbol': 'amzn', 'price': 840.05}] AS t
```

results into the tuple `{'tdc':31.52, 'amzn':840.05}`.

The treatment of `MISSING` is same to the treatment of `MISSING` by `SELECT VALUE` (Section refsec:tuple-constructor). Namely, whenever an attribute name or attribute value evaluates to `MISSING`, the corresponding attribute name/value pair will not appear in the tuple.

Example 23. The query

```
PIVOT t.price AT t.symbol
FROM [{'symbol':25, 'price':31.52}, {'symbol':'amzn', 'price':840.05}] AS t
```

results into the tuple `{'amzn': 840.05}` since `25` is not a legitimate attribute name.

6.3 SQL SELECT list as Syntactic Sugar of SELECT VALUE

6.3.1 SELECT Without *

The SQL syntax:

```
SELECT e_1 AS a_1, ..., e_n AS a_n
```

is syntactic sugar for:

```
SELECT VALUE {a'_1:e_1, ..., a'_n:e_n}
```

whereas if the attribute name a_i is written as an identifier (e.g., `a` or `"a"`) it is replaced by a single-quoted form a'_i (e.g., `'a'`).

When the expression e_i is of the form $e'_i.n$ (i.e. a path that navigates into tuple attribute n), PartiQL follows SQL in allowing the attribute name to be optional. In this case,

```
SELECT ... e_i.n ...
```

is equivalent to

```
SELECT ... e_i.n ... AS n
```

In the case that the expression e_i is not of the form $e'_i.n$ the clause:

```
SELECT ... e_i ...
```

is equivalent to

```
SELECT ... e_i AS a_i ...
```

where a_i is a system-generated name. SQL and PartiQL do not provide a standard convention.

6.3.2 SQL's *

Consider a query whose FROM defines a variable x that has no schema and the SELECT clause includes at least one x.*. Let us first consider the simpler case where the SELECT clause is a single item x.*. Then the clause

```
SELECT x.*
```

reduces to

```
SELECT VALUE CASE WHEN NOT x IS TUPLE THEN {'_1': x} ELSE x END
```

Notice that PartiQL extends the .* to also operate on x bindings that are not tuples. These are converted to singleton tuples with a synthetic name.

Example 24. The query

```
SELECT x.*
FROM [{'a':1, 'b':1}, {'a':2}, 'foo'] AS x
```

results into `<< {'a':1, 'b':1}, {'a':2}, {'_1':'foo'} >>`. Notice that the input has a non-tuple that was converted to a tuple with a synthetic attribute name `_1`, this is because the result of a traditional SELECT is always a container of tuples.

We generalize the semantics of a SELECT list, where at least one of the items is a .* item, we use the function TUPLEUNION. When all of $t_1, t_2, \ldots, t_n$ are tuples TUPLEUNION$(t_1, t_2, \ldots, t_n)$ outputs a tuple t such that for each attribute name/value pair $n : v$ of any t_i, the tuple t has a respective $n : v$. Notice the possibility that the output t has duplicate attribute names because either (i) two different inputs t_i and t_j had the same attribute name, or (ii) because an input t_i already had a duplicate attribute name.

Using TUPLEUNION, we rewrite the SELECT clause as illustrated by the following example, which has two .* items and one conventional item. The generalization to more items, of either kind should be obvious. Notice that if v_1 (resp. v_3) is bound to a non-tuple value v, then it is treated as if it were the tuple {'_1':v_1} (resp. {'_2':v_3}.

```
SELECT v_1.*, e_2 AS a, v_3.*
```

is equivalent to

```
SELECT VALUE TUPLEUNION(
    CASE WHEN v_1 IS TUPLE THEN v_1 ELSE {'_1': v_1} END,
    {'a':e_2},
    CASE WHEN v_3 IS TUPLE THEN v_3 ELSE {'_2': v_3} END
)
```

Notice that the attribute names `_1`, `_2` have been invented.

6.4 Examples with combinations of multiple features

Example 25. A SFW subquery may appear in the `SELECT VALUE` clause of a query, enabling the creation of nested results.

Consider the database

```
sensors :  [{'sensor':1},
            {'sensor':2}
            ]
logs:  [{'sensor':1, 'co':0.4},
        {'sensor':1, 'co':0.2},
        {'sensor':2, 'co':0.3}
        ]
```

The query

```
SELECT VALUE {'sensor':  s.sensor,
              'readings':  (SELECT VALUE l.co
                            FROM logs AS l
                            WHERE l.sensor = s.sensor
                            )
             }
FROM sensors AS s
```

results into

```
≪{'sensor':1, 'readings':≪0.4, 0.2 ≫},
 {'sensor':2, 'readings':≪0.3 ≫}
≫
```

Notice that each tuple of the result has a nested array, which has been created by the inner `SELECT VALUE`.

The query could also have been written using `SELECT` (instead of `SELECT VALUE`) for the outer query, as follows:

```
SELECT s.sensor AS sensor,
       (SELECT VALUE l.co
        FROM logs AS l
        WHERE l.sensor = s.sensor
        ) AS readings
FROM sensors AS s
```

Furthermore, the `AS sensor` could be ommitted (as in SQL).

Example 26. This example shows how the combined action of `UNPIVOT` and `PIVOT` enables to analyze the attribute names. Consider the following database that has a sequence of measurements of various gases.

```
sensors :  [{'no2':0.6, 'co':0.7, 'co2':0.5},
            {'no2':0.5, 'co':0.4, 'co2':1.3}
            ]
```

The following query keeps only the carbon oxides. [3]

```
SELECT VALUE (PIVOT v AT g
              FROM UNPIVOT r AS v AT g
              WHERE g LIKE 'co%')
FROM sensors AS r
```

The result is

```
[{'co':0.7, 'co2':0.5},
 {'co':0.4, 'co2':1.3}
 ]
```

Intuitively, the `UNPIVOT` turns every instance of the tuple `t` into a collection. The `WHERE` filters the collections. The `PIVOT` pivots the filtered collections back into tuples.

[3] The query author is pretty weak in chemistry and cannot enumerate the carbon oxides explicitly in her query.

7 Functions

The semantics of predicates (i.e., functions returning booleans) and (non-aggregate) functions in PartiQL are identical to those of SQL when their inputs are those that are allowed by SQL. PartiQL makes the following extensions for the cases where the inputs are beyond those allowed by SQL.

7.1 Inputs with wrong types:

Unlike SQL where typing issues can be detected during query compilation, the permissive option of PartiQL has to define semantics for the case where the inputs of a function are not compatible with the function/predicate arguments. Furthermore, PartiQL facilitates propagating missing input attributes to respective missing output attributes.

Alike SQL, all functions have input argument types that they conform to. For example, the function `log` expects numbers. All functions return `MISSING` when they input data whose types do not conform to the input argument types. Since no function (other than `IS MISSING`) has `MISSING` as an input argument type, it follows that all functions return `MISSING` when one of their inputs is `MISSING`.

Example 27. The query

```
SELECT VALUE {'a':3*v.a, 'b':3*(CAST v.b AS INTEGER)}
FROM [{'a':1, 'b':'1'}, {'a':2}] v
```

results into `≪{'a':3, 'b':3}, {'a':6} ≫`. Notice how the missing `b` attribute in the input leads to a respective missing attribute in the output.

Example 28. Each one of these expressions returns `MISSING`: `5 + missing`, `5 > 'a'`, `NOT {a:1}`.

7.1.1 Equality

Equality never fails in the type-checking mode and never returns `MISSING` in the permissive mode. Instead, it can compare values of any two types, according to the rules of the PartiQL type system. For example, `5 = 'a'` is `false`.

Since PartiQL variables may bind to composite values (collections, tuples), PartiQL extends the semantics of equality for these cases. In particular, equality in PartiQL is *deep equality*, defined as follows:

1. Given two arrays x and y that have the same length l, the result of $x = y$ is the result of

   $$\texttt{eqg}(x[0], y[0])\ \texttt{AND}\ \ldots\ \texttt{AND}\ \texttt{eqg}(x[l], y[l])$$

   The `eqg`, unlike the `=`, returns true when a `NULL` is compared to a `NULL` or a `MISSING` to a `MISSING`. When the arrays x and y do not have the same length, the $x = y$ is `false`.

2. A similar straightforward equality applies to tuples: They have to have the same attributes. Then equality $t_1 = t_2$ is true if

   $$\texttt{eqg}(t_1.a, t_2.a_1)\ \texttt{AND}\ \ldots\ \texttt{AND}\ \texttt{eqg}(t_1.a_n, t_2.a_n)$$

   where $a_1, \ldots, a_n$ are the attributes that appear in t_1 and t_2.

3. Equality for bags is similarly straightforward: Two bags x and y are equal if and only if every element e of x that appears n times in x also appears n times in y.

Example 29. The following are true:

```
≪3, 2, 4, 2 ≫= ≪2, 2, 3, 4 ≫
{'a':1, 'b':2} = {'b':2, 'a':1}
{'a':[0,1], 'b':2} = {'b':2, 'a':[0,1]}
```

The following are false:

```
≪3, 4, 2 ≫= ≪2, 2, 3, 4 ≫
{'a':1, 'b':2} = {'a':1}
{'a':[0,1], 'b':2} = {'b':2, 'a':[0,1,2]}
```

The following are also false.

```
{'a':1, 'b':2} = {'a':1}
{'a':1, 'b':2} = {'a':1, 'b':null}
{'a':[0,1], 'b':2} = {'b':2, 'a':[null,1]}
```

8 WHERE clause

The **WHERE** clause inputs the bindings that have been produced from the **FROM** clause and outputs the ones that satisfy its condition.

The boolean predicates follow SQL's 3-valued logic. Recall, PartiQL has two kinds of absent values: **NULL** and **MISSING**. As far as the boolean connectives and **IS NULL** are concerned a **NULL** input and a **MISSING** input behave identically. For example, **MISSING AND TRUE** is equivalent to **NULL AND TRUE**: they both result into **NULL**.

For the semantics of equality and of other functions, see Section 7.

Alike SQL, when the expression of the **WHERE** clause expression evaluates to an absent value or a value that is not a Boolean, PartiQL eliminates the corresponding binding.

Example 30. The result of

```
SELECT VALUES v.a
FROM [{'a':1, 'b':true}, {'a':2, 'b':null}, {'a':3}] v
WHERE v.b
```

is `≪1 ≫`.

The predicate **IS MISSING** allows distinguishing between **NULL** and **MISSING**: **NULL IS MISSING** results to false; **MISSING IS MISSING** results to true.

9 Coercion of subqueries

In PartiQL, as is the case with SQL as well, expressions may involve SFW subqueries (Figure 3, line 20). PartiQL SFW subqueries are enclosed in parentheses (i.e., identical to SQL). For compatibility with SQL, a SFW subquery starting with a `SELECT` clause (as opposed to a subquery starting with `SELECT VALUE` or `PIVOT`) coerces into a scalar or into an array, depending on the context. The following cases replicate SQL's coercing behavior and analyze in which cases the result of a subquery coerces into scalar and in which cases they coerce into arrays.

An PartiQL extension with respect to SQL is that, in the permissive mode, subqueries that fail to coerce to the required type (scalar or tuple) still run, as opposed to failing. They simply omit from the results the data that correspond to the coercion failures.

9.1 Coercion of a `SELECT` subquery into a scalar

In each of the following cases a SFW subquery coerces into a scalar

- if it appears as the rhs of a comparison operator (`=`, `>`, etc) where the lhs is not an array literal. And, vice versa, if it appears as the lhs of a comparison operator where the rhs is not an array literal. (If it is the lhs of a comparison operator where the lhs is an array literal, it coerces into array, per Section 9.2.)
- if it is an SFW subquery expression that (a) is not the collection expression of a `FROM` clause item and (b) is not the rhs of an `IN`. (If it is the rhs of an `IN` then it should not be coerced; see note on semantics of `IN`, Section `??`.)

Essentially, a subquery that is coerced may appear in all clauses except the `FROM`. For example, it may be a `SELECT` subquery s that appears as an item of a `SELECT`, `SELECT VALUE` or `PIVOT` clause. Or it may be a subexpression of an expression that appears in `SELECT`, `SELECT VALUE` or `PIVOT` clause. Or it may be a subexpression of the `WHERE` clause expression, as long as it is not the rhs of an `IN`. In any of these cases the result of the subquery s is cast into a scalar.

Technically, the subquery s (which uses `SELECT`) is rewritten into an equivalent subquery s' that utilizes `SELECT VALUE`, by following the steps of Section 6.3. Then the result of s' is cast into a scalar by applying the function `COLL_TO_SCALAR(`s'`)`.

Example 31. The SQL query

```
SELECT v.foo,
       (SELECT w.bar
        FROM someDataSet w
        WHERE w.sth = v.sthelse) AS bar
FROM anotherDataSet v
```

is rewritten into

```
SELECT VALUE {'foo':  v.foo
              'bar':  COLL_TO_SCALAR(SELECT VALUE {'bar':  w.bar}
                                     FROM someDataSet w
                                     WHERE w.sth = v.sthelse)}
FROM anotherDataSet v
```

As is the common semantics of PartiQL in the permissive mode, when `COLL_TO_SCALAR` fails to cast the subquery into a scalar, it outputs `MISSING`. The inputs that are coerced into scalars are the ones that SQL prescribes: When the input is a collection consisting of a single tuple with a single attribute, the input is coerced into a scalar. All other inputs to `COLL_TO_SCALAR` lead to `MISSING`.

Example 32. In this example, in one instance the inner `SELECT` evaluates to a collection with more than one element. Because the `COLL_TO_SCALAR` function produces a `MISSING` instead of failing, the query works.

Consider the tables

```
customers :  [{'id':1, 'name':'Mary'},
              {'id':2, 'name':'Helen'},
              {'id':1, 'name':'John'}
              ]
orders :  [{'custId':1, 'name':'foo'},
           {'custId':2, 'name':'bar'}
           ]
```

The following query would fail in SQL, because there are two customer tuples with the same id. Of course, in a well-designed SQL database that has a primary key or uniqueness constraint on the id, there would not be two customers with the same id. However, lack of constraints is typical in the data targeted by PartiQL. This query runs in the permissive mode of PartiQL.

```
SELECT o.name AS orderName,
       (SELECT c.name FROM customers c WHERE c.id=o.custId) AS customerName
FROM orders o
```

The result is

```
≪ {'orderName':'foo'}, {'orderName':'bar', 'customerName':'Helen'} ≫
```

Notice the missing `'customerName'` in the first tuple.

As in SQL, an implementation with static type checks will be able to detect and warn that, in certain cases, a coercion will always fail and produce `missing`.

Example 33. The following `SELECT` clause is guaranteed to produce tuples with `bar1` and `bar2`. Thus it cannot coerce into scalar.

```
SELECT w.bar1 AS bar1, w.bar2 AS bar2
FROM someDataSet w
```

Static type analysis can infer that the nested query above will deliver tuples consisting of `bar1` and `bar2`. Thus, even before accessing any data, it can warn the user that this query is erroneous.

9.2 Coercion of a SELECT subquery into an array

An `SELECT` SFW subquery coerces into an array when it is the rhs (respectively, lhs) of a comparison operator whose other argument is an array.
[4]

The reduction of a `SELECT` subquery to the PartiQL is exhibited by the following example.

Example 34. The SQL query

```
SELECT v.foo
FROM anotherDataSet v
WHERE (v.a, v.b) = (SELECT w.c, w,d
                    FROM someDataSet w
                    WHERE w.sth = v.sthelse)
```

is rewritten into

```
SELECT VALUE {'foo':  v.foo}
FROM anotherDataSet v
WHERE (v.a, v.b) = (SELECT VALUE [w.c, w,d]
                    FROM someDataSet w
                    WHERE w.sth = v.sthelse)
```

[4] Recall, in the interest of compatibility to SQL, PartiQL allows array literals to be denoted with parentheses instead of brackets (see Section 6.1.2).

10 Scoping rules

As far as the variables environment is concerned, the scoping rules are identical to those of SQL. Section 3.4 explained how the resolution of variable naming conflicts favors the variables defined by the inner queries.

The scoping rules discussed in the present section discuss the resolution of naming conflicts between names defined in the database environment and the variables of the environment variables. The potential for such naming conflicts is driven by the nested data of PartiQL, as illustrated next.

Notice there are a few more naming conventions, pertaining to the use of attribute names defined in the **SELECT** clause into the **GROUP BY** and **ORDER BY** clause. These conventions are explained in along with the semantics of the respective clauses (see Sections 11 and 12).

Example 35. The following example illustrates how SQL compatibility issues and the needs of navigating into nested data need to be carefully merged together. Consider the following database that has a table **c**, i.e. a collection of tuples, and also named data **x.n** and **y**.

```
t.c:  ≪ {'a':1, 'n':[{'b':11, 'c':12}]},
        {'a':2, 'n':[{'b':21, 'c':22}]}
        ≫
x.n :  ≪ {'b':3} ≫
y:  {'a':1, 'b':2}
```

Then consider the query

```
SELECT t.a
FROM t.c AS x
WHERE x.a IN (SELECT y.b FROM x.n AS y)
```

This query poses many scoping issues:

1. Does **x.n** refer to the named value **x.n** or to the **n** attribute of the variable **x**? For SQL compatibility purposes it refers to the named value **x.n**. Read below how to refer to the variable **x**.

2. Does **y.b** refer to the **b** attribute of the **y** attribute or to the **b** attribute of the named value **y**? For SQL compatibility purposes it refers to the **b** attribute of the variable **y**.

Notice how SQL compatibility required the database environment to take priority over the variables environment in the **FROM** clause and then, vice versa, the variables environment to take priority over the database environment in the **SELECT** clause.

Scoping rules resolving naming conflicts between variables and database names: Since the rules are easier to express when all database names are a single identifier, such as **thedb** or **"the db"** (as opposed to paths, such as **somedb.sometable**), we first specify the scoping rules under the assumption that all database names are a single identifier. We remove the assumption and generalize later.

In the absence of schema the following rules apply

1. **@***identifier* refers to the environment variable named *identifier*; if there is no such environment variable, the *identifier* refers to the database name *identifier*; if there is no such database name either, the query fails compilation.

2. in a **FROM** clause path that starts with *identifier*, the *identifier* refers to the database name *identifier*; if there is no such database name, the *identifier* refers to a variable; otherwise query fails compilation. [5]

3. in a non-**FROM** clause path that starts with *identifier*, the *identifier* refers to the environment variable named *identifier*; if there is no such environment variable, the *identifier* refers to the database name *identifier*; if there is no such database name either, the query fails compilation.

Next, we generalize to also allow for the possibility of database names of the form *identifier*.*indentifier*.... The following rules apply regarding the semantics of $i_1.i_2.\ldots.i_n$, where $i_1, i_2, \ldots i_n$ are identifiers.

- **@**$i_1.i_2.\ldots.i_n$ always refers to the environment variable named i_1; if there is no such variable and $i_1.i_2.\ldots.i_m, m \leq n$ is a database name then $i_1.i_2.\ldots.i_m$ refers to such named database name. Again, if there is a choice, choose the largest m. If both the resolution to variable and the resolution to database name, fail the query during compilation.

[5] A path is a **FROM** clause path if it appears in the **FROM** clause of the SFW query in which it is *immediately* nested.

- if $i_1.i_2.\ldots.i_n$ is a **FROM** path and $i_1.i_2.\ldots.i_m, m \leq n$ is a database name then $i_1.i_2.\ldots.i_m$ refers to such named database name and $i_{m+1}.\ldots.i_n$ is a series of tuple path navigations starting from the database name $i_1.i_2.\ldots.i_m$. If there is a choice, choose the largest m, i.e., the longest database name.
- if $i_1.i_2.\ldots.i_n$ is a non-**FROM** clause expression and i_1 is an environment variable then i_1 refers to such variable; if there is no such variable and $i_1.i_2.\ldots.i_m, m \leq n$ is a database name then $i_1.i_2.\ldots.i_m$ refers to such named database name. Again, if there is a choice, choose the largest m. If both the resolution to variable and the resolution to database name, fail the query during compilation.

Example 36. Assume database names `coll`, `v.foo`, `w`. Then in the query

```
1 SELECT v.foo
2 FROM coll AS v, @v.foo AS w,
3           (SELECT w.a, u.b FROM @w.bar AS u)
4           AS x
```

`coll` refers to the database name. The `v` in `@v.foo` refers to the variable `v`. If the `@` were not there, `v.foo` would refer to the database name `v.foo`. The `w` in `w.a` refers to the variable defined in line 2.

Note, the expressions `coll` and `@v.foo` are **FROM** clause expressions because they appear in the **FROM** clause of the *sfw_query* of lines 1-4, in which they are immediately nested. Similarly, the expression `@w.bar` is a **FROM** clause expression because it appears in the **FROM** clause of the *sfw_query* of line 3, in which it is immediately nested. In contrast, the expressions `w.a` and `u.b` are not **FROM** clause expressions. Though they are nested into the **FROM** clause of the query of lines 1-4, they are not immediately nested into the query of lines 1-4.

11 GROUP BY clause

The PartiQL **GROUP BY** clause expands SQL's grouping. Unlike SQL, the PartiQL **GROUP BY** can be thought of as a standalone operator that inputs a collection of binding tuples and outputs a collection of binding tuples.

As is typical in many clauses, the semantics proceed in two steps:

- Section 11.1 explains the core PartiQL **GROUP BY** structure.
- Section 11.2 shows that SQL's **GROUP BY** can be explained over the core **GROUP BY**.

11.1 PartiQL GROUP BY core: Grouping into a Group Variable

The **GROUP BY** clause (Figure 3, lines 9–11)

$$\texttt{GROUP BY } e_1 \texttt{ AS } x_1 \texttt{ , } \ldots \texttt{ , } e_m \texttt{ AS } x_m \texttt{ GROUP AS } g$$

creates a group. Each e_i is a *grouping expression*, each x_i is a *grouping variable* [6] and g is the *group variable*.

As in SQL, the bag of input binding tuples $B^{in}_{\texttt{GROUP}}$ is partitioned into the minimal number of equivalence groups $B_1 \ldots B_n$, such that any two binding tuples $b, b' \in B^{in}_{\texttt{GROUP}}$ are in the same equivalence group if and only if every grouping expression e_i evaluates to equivalent values v_i (when evaluated on b) and v'_i (when evaluated on b'). More precisely, as in SQL, there is an equivalence function **eqg**, used by the **GROUP BY** to determine if two values v_i and v'_i are equivalent for grouping purposes. The equivalence function $\texttt{eqg}(v_i, v'_i)$ returns only true or false; true meaning that the values are equivalent for grouping purposes. See Section 11.1.1 for specifics of **eqg**. If a grouping expression evaluates to **MISSING**, it is first coerced into **NULL**, thus bringing **MISSING** and **NULL** in the same group.

Unlike SQL, for each group B_j $(1 \leq j \leq n)$, the **GROUP BY** clause outputs a binding tuple $b_j = \langle x_1 : v_1,\ \ldots,\ x_m : v_m,\ g : B_j \rangle$ that has the full group B_j. Notice:

1. the binding tuples that appear in the g collection have one attribute for each of the variables defined in the **FROM** clause, since these binding tuples come as-is from $B^{in}_{\texttt{GROUP}}$.
2. even if the bag $B^{in}_{\texttt{GROUP}}$ is flat binding tuples, the output bag $B^{out}_{\texttt{GROUP}}$ is not just flat binding tuples, since g has nested binding tuples. Note, we have been explicitly denoting binding attributes with **MISSING** values in the binding tuples. However, once these binding tuples become the tuples of the PartiQL data model, any binding attribute with **MISSING** value will not appear.

Example 37. Consider again the **logs** data of Example 25 and assume that we want to group the **co** readings by sensor. The following query solves the problem using only core features.

```
SELECT VALUE {'sensor':  sensor,
              'readings':  (SELECT VALUE v.l.co FROM g AS v) }
FROM logs AS l
GROUP BY l.sensor AS sensor GROUP AS g
```

The **GROUP BY** outputs the collection of binding tuples

$B^{out}_{\texttt{GROUP}} = B^{in}_{\texttt{SELECT}} =$
```
≪⟨ sensor:1, g:  ≪⟨l:{'sensor':1, 'co':0.4}⟩,
                  ⟨l:{'sensor':1, 'co':0.2}⟩
                  ≫⟩,
  ⟨ sensor:2, g:≪⟨l:{'sensor':2, 'co':0.3}⟩ ≫⟩
 ≫
```

Notice that the collection **g** has tuples with a single attribute **l**, since this is the single variable of the **FROM** clause in this example.

Consequently the **SELECT** clause outputs

```
≪{'sensor':1, 'readings':≪0.4, 0.2≫},
 {'sensor':2, 'readings':≪0.3 ≫}
≫
```

[6] Grouping variables is an extension of SQL by PartiQL, which interestingly simplifies dramatically the explanation of SQL semantics, as it enables the **GROUP BY** to be seen as a standalone function.

Notice that the query of Example 25 and the query of the present example do not always produce the same result. For example, if there were no readings for a sensor, the query of Example 25 would still have this sensor in the result (and its `readings` would be empty). In contrast, the query of the present example will not have this sensor in the result.

Here is a shorter equivalent query that uses PartiQL collection paths and SQL's aliases.

```
SELECT VALUE {'sensor':  sensor,
              'readings':  g[*].l.co }
FROM logs AS l
GROUP BY l.sensor AS sensor GROUP AS g
```

Notice, the output binding tuple provides the partitioned input binding tuples in the group variable g, which can be explicitly utilized in subsequent `HAVING`, `ORDER BY` and `SELECT` clauses. Thus, an PartiQL query can perform complex computations on the groups, leading to results of any type (e.g. collections nested within collections). The explicit presence of groups in PartiQL, while more general than SQL, also leads to simpler semantics than those of SQL, since the `GROUP BY` clause semantics are independent of the presence of subsequent functions in `HAVING`, `ORDER BY` and `SELECT`.

Example 38. The following PartiQL query counts and averages the readings of each sensor. It also refers to the `logs` of Example 25. The `COLL_COUNT` function is simply given the group variable and counts how many elements are in that collection.

```
SELECT VALUE {'sensor':  sensor,
              'avg':  COLL_AVG(SELECT VALUE v.l.co FROM g AS v),
              'count':  COLL_COUNT(g) }
FROM logs AS l
GROUP BY l.sensor AS sensor GROUP AS g
```

Notice, the aggregate functions `COLL_AVG` and `COLL_COUNT` (and for that matter, by convention, any function starting with `COLL`) can be thought of as general-purpose functions. Generally, they do not have to be fed by the result of a grouping operation - unlike SQL's `COUNT` and `AVG` that are being fed exclusively from the results of grouping operations. (Furthermore, the SQL `COUNT` and `AVG` make use of SQL's syntactic sugar, where there is no explicit use of group variable, as explained in Section 11.2.2.)

Example 39. This is a legitimate PartiQL expression:

```
COLL_COUNT([5, {a:2, b:3}])
```

The result is `2`, since the input to `COLL_COUNT` is an array with two elements.

Similarly, it is fine to include in any clause an aggregate function fed by the result of a (sub)query.

Example 40. In the following expression `COLL_COUNT` inputs the result of a query

```
COLL_COUNT(SELECT VALUE x FROM logs x WHERE x.sensor=1)
```

Remark: An efficient implementation will often avoid materializing the group variable. In many cases, like the ones of the above examples, the group can be streamed into the aggregate function.
Remark: The semantics of the `SELECT` and `HAVING` clauses do not need to be aware of the presence of `GROUP BY` and treat differently (as SQL would do) these classes of functions:

- scalar functions (e.g. `+`) that input scalars and output scalars
- SQL aggregation functions (e.g. `SUM`) that input bags and output scalars

Indeed, `HAVING` behaves identical to a `WHERE`, once groups are already formulated earlier.

The PartiQL approach provides two benefits: First, it leads to shorter, modular semantics. Second, it enables `GROUPY` to address use cases that would otherwise need knowledge and non-trivial SQL programming of window functions. See Example 47.

11.1.1 Equivalence function used by grouping; grouping of NULL and MISSING

The equivalence function **eqg** extends SQL's respective function. In particular, it behaves as follows:

- **eqg**(NULL, NULL) is true, despite NULL = NULL not being true.
- for any two non-null values x and y, **eqg**(x, y) returns the same with $x = y$. As is the case generally for =, while SQL's = will error when given incompatible types, while the PartiQL = will return **false**.

Notice that PartiQL will group together the NULL and the MISSING grouping expressions, since any grouping expression resulting to MISSING has been coerced into NULL before **eqg** does comparisons for grouping. Example 41 shows the repercussions of coercing NULL into MISSING and also shows how to discriminate between NULL and MISSING, if so desired.

Example 41. The query of Example 37 will group together any log readings where the **sensor** attribute is either NULL or is altogether MISSING. For example, if **logs** is

```
logs:[ {'sensor':  1, 'co':0.4},
       {'sensor':  2, 'co':0.3},
       {'sensor':  null, 'co':0.1},
       {'sensor':  1, 'co':0.2},
       {'co':0.5}
     ]
```

then the **GROUP BY** will output the collection of binding tuples

$B^{out}_{\texttt{GROUP}} = B^{in}_{\texttt{SELECT}}$ =
```
≪⟨ sensor:1, g:  ≪⟨l:{'sensor':1, 'co':0.4}⟩,
                 ⟨l:{'sensor':1, 'co':0.2}⟩
                 ≫⟩,
  ⟨ sensor:2, g:≪⟨l:{'sensor':2, 'co':0.3}⟩ ≫⟩,
  ⟨ sensor:null, g:  ≪⟨l:{'sensor':null, 'co':0.1}⟩,
                     ⟨l:{'co':0.5}⟩
                     ≫⟩
≫
```

Notice that both the 3rd and 5th tuples of **logs** were grouped under the **sensor:null** group, despite the **sensor** of the 3rd being NULL while the **sensor** of the 5th being MISSING. The query result is

```
≪{'sensor':1, 'readings':≪0.4, 0.2≫},
 {'sensor':2, 'readings':≪0.3≫},
 {'sensor':null, 'readings':≪0.1, 0.5≫}
≫
```

If we wanted to discriminate the NULL from the MISSING we could write the following query

```
SELECT VALUE {'sensor':  CASE WHEN missingFlag THEN MISSING ELSE sensor END,
              'readings':  (SELECT VALUE v.l.co FROM g AS v) }
FROM logs AS l
GROUP BY l.sensor IS MISSING AS missingFlag, l.sensor AS sensor GROUP AS g
```

In this case the **GROUP BY** would output the collection of binding tuples

$B^{out}_{\texttt{GROUP}} = B^{in}_{\texttt{SELECT}}$ =
```
≪⟨missingFlag:false,
  sensor:1, g:  ≪⟨l:{'sensor':1, 'co':0.4}⟩,
                ⟨l:{'sensor':1, 'co':0.2}⟩
  ≫⟩,
 ⟨ missingFlag:  false, sensor:2, g:≪⟨l:{'sensor':2, 'co':0.3}⟩ ≫⟩
 ⟨ missingFlag:  false, sensor:null, g:≪⟨l:{'sensor':null, 'co':0.1}⟩ ≫⟩
 ⟨ missingFlag:  true, sensor:null, g:≪⟨l:{'co':0.5}⟩ ≫⟩
≫
```

and the query result would be

```
≪{'sensor':1, 'readings':≪0.4, 0.2≫},
 {'sensor':2, 'readings':≪0.3≫},
 {'sensor':null, 'readings':≪0.1≫},
 {'readings':≪0.5≫}
≫
```

11.1.2 The GROUP ALL variant

The **GROUP ALL** variant of **GROUP BY** outputs a single binding tuple, regardless of whether the **FROM**/**WHERE** produced any tuples, i.e., regardless of whether its input $B^{in}_{\texttt{GROUP}}$ is empty or not.

The **GROUP ALL** is not increasing the expressiveness of PartiQL. Example 42 shows how to achieve without **GROUP ALL**, what the **GROUP ALL** can do. However, we include **GROUP ALL** for facilitating the reduction of SQL's aggregation into the core PartiQL (see Section 11.2.2).

Example 42. Consider again the **logs** data of Example 25 and assume that we want to count the total number of readings that are above **1.5** with a core PartiQL query. (Example 45 does the same with SQL.)

```
SELECT VALUE {'largeco':  COLL_COUNT(g)}
FROM logs AS l
WHERE l.co > 1.5
GROUP ALL AS g
```

Notice, there are no readings above **1.5** in the example data. Since there is no tuple that satisfies the **WHERE** clause

$$B^{out}_{\texttt{WHERE}} = B^{in}_{\texttt{GROUP}} = \ll \gg$$
$$B^{out}_{\texttt{GROUP}} = B^{in}_{\texttt{SELECT}} = \ll \langle \texttt{g} : \ll \gg \rangle \gg$$

Since **COLL_COUNT(** $\ll \gg$ **)** is **0**, the query result is the collection

$$\ll \texttt{\{'largeco': 0 \}} \gg$$

Therefore the PartiQL query is equivalent to the plain SQL query

```
SELECT COUNT(*) AS largeco
FROM logs AS l
WHERE l.co > 1.5
```

The following core PartiQL also accomplishes the same computation, without using **GROUP ALL**.

```
{'largeco':  COLL_COUNT(SELECT VALUE l
                        FROM logs AS l
                        WHERE l.co > 1.5
                        )
}
```

11.2 SQL compatibility features

The group-by and aggregation of PartiQL is backwards compatible to SQL.

11.2.1 Grouping Attributes and Direct Use of Grouping Expressions

For SQL compatibility PartiQL allows

GROUP BY $\ldots, e, \ldots$

i.e., a grouping expression e that is not associated with a grouping variable x. (In core PartiQL, one would write e **AS** x.)

For SQL compatibility, PartiQL supports using the grouping expression e in **HAVING**, **ORDER BY** and **SELECT** clauses. The SQL form (S) is syntactic sugar for the core PartiQL (C).

(S)		(C)	
FROM	$\ldots$	**FROM**	$\ldots$
GROUP BY	$e, \ldots$	**GROUP BY**	e **AS** $x, \ldots$
HAVING	$f(e, \ldots)$	**HAVING**	$f(x, \ldots)$
ORDER BY	$f'(e), \ldots$	**ORDER BY**	$f'(x), \ldots$
SELECT	$f''(e), \ldots$	**SELECT**	$f''(x), \ldots$

Example 43. The SQL-compatible query

```
SELECT v.a+1 AS bar
FROM foo AS v
GROUP BY v.a+1
```

is written in core PartiQL as

```
SELECT VALUE {'bar':  x}
FROM foo AS v
GROUP BY v.a+1 AS x GROUP AS dontcare
```

Remark: What is "same expression"?: An open question in the equivalence of (C) and (S) is the exact meaning of "same expression e in `GROUP BY` and `SELECT` (or `HAVING`, `ORDER BY`)". Is `v.a + 1` the same with `1 + v.a`? Is `v.a + 1` the same with `a + 1` in the presence of a schema that dictates that the variable `v` is a tuple with an attribute `a`? Both SQL and PartiQL answer "no" and "yes" respectively to the two questions. In particular:

An expression e that appears in the `GROUP BY` clause and an expression e' that appears in the `SELECT` or `HAVING` or `ORDER BY` are considered the same expression if they are syntactically identical after performing the schema-based rewritings of Section 15.

11.2.2 SQL's Implicit Use of the Group Variable in SQL Aggregate Functions

SQL does not have explicit group variables. For SQL compatibility, PartiQL allows the SQL aggregation functions to be fed by expressions that do not explicitly say that there is iteration over the group variable. Suppose that a query

1. is a `SELECT` query,
2. lacks a `GROUP AS` clause, and
3. any of the `SELECT`, `HAVING` and/or `ORDER BY` clauses contains a function call $f(e)$, where f is a *SQL aggregation function* such as `SUM` and `AVG`. (See Section 11.2.3

Then, the query is rewritten as follows:

- if the query has a `GROUP BY` clause, add to it

  `GROUP AS` g

  where g is a fresh variable, i.e., a variable that is not a database name nor a variable of the query or a variable of the queries within which it is nested.

- if the query has no `GROUP BY` clause, add to it

  `GROUP ALL GROUP AS` g

  where g is a fresh variable.

- if the aggregation function call is `COUNT(*)`, then rewrite into `COUNT`(g)

- otherwise, rewrite $f(e)$ into

$$f(\texttt{SELECT VALUE } e' \texttt{ FROM } g \texttt{ AS } p)$$

  where e' is produced from e as follows: Consider the variables $v_1, \ldots, v_n$ that appear in $B^{in}_{\texttt{GROUP}}$ (i.e., the variables defined by the query's `FROM` and `LET` clauses) and are not grouping attributes. Substitute each identifier v_i (that does not stand for attribute name) in e with $p.v_i$.

Example 44. Consider again the query of Example 38. It can be written in an SQL compatible way as

```
SELECT l.sensor AS sensor,
       AVG(l.co) AS avg,
       COUNT(*) AS count
FROM logs AS l
GROUP BY l.sensor
```

Example 45. The query of Example 42 can be written in standard SQL syntax as

```
SELECT COUNT(g) AS largeco
FROM logs AS l
WHERE l.co > 1.5
```

Notice that SQL does not allow nested aggregate functions. Respectively, PartiQL does not allow one to write queries that lack a `GROUP AS` or `GROUP ALL` clause and have nested aggregate SQL functions.

11.2.3 Designation of SQL aggregate functions

Each implementation will have a list of SQL aggregate functions, which are not necessarily just the ones prescribed by the standard (`COUNT`, `SUM`, `AVG`, etc). (Recall from Section 11.2.2 that SQL aggregate functions do not use an explicit group variable.)

Furthermore, it is required that for each SQL aggregate function f, if an implementation offers a corresponding core PartiQL aggregate function, the PartiQL function is named `COLL_`f. For example, the core PartiQL aggregate `COLL_AVG` corresponds to the SQL aggregate `AVG`. Nevertheless, it is possible that an implementation offers only `COLL_AVG` or offers only `AVG`. The semantic relationship between the SQL aggregate function and the corresponding core PartiQL aggregate function is the one explained in Section 11.2.2: The SQL aggregate functions do not input explicit group variables and, thus, their semantics are explained by the reduction to the corresponding core PartiQL aggregate.

11.2.4 Aliases from `SELECT` clause

In SQL, a grouping expression may be an alias that is defined by the `SELECT` clause. For compatibility purposes, PartiQL adopts the same behavior.

The query (S), which uses the `SELECT`-defined alias feature, is syntactic sugar for the query (C). Notice that the grouping expression a is simply a shorthand for e.

(S) `SELECT` ...,e `AS` a,...
`FROM` ...
`GROUP BY` ...,a,...

(C) `SELECT` ...,e `AS` a,...
`FROM` ...
`GROUP BY` ...,e,...

In the case that the grouping expression is a constant positive integer literal n, then it stands for the nth attribute of the `SELECT` clause. However, this requires that the tuples produced by the `SELECT` have schema and they are ordered tuples. The relevant examples will be provided in the schema section.

Example 46. Consider the database

```
people: ≪{'name': 'zoe', 'age': 10, 'tag': 'child'},
          {'name': 'zoe', 'age': 20, 'tag': 'adult'},
          {'name': 'bill', 'age': 30, 'tag': 'adult'}
          ≫
```

The query

```
SELECT p.tag || ':' || p.name AS tagname, AVG(p.age) AS average
FROM people AS p
GROUP BY tagname
```

is equivalent to the query

```
SELECT p.tag || ':' || p.name AS tagname, AVG(p.age) AS average
FROM people AS p
GROUP BY p.tag || ':' || p.name
```

Either query results into

```
people: ≪{'tagname': 'child:zoe', 'average': 10},
          {'tagname': 'adult:zoe', 'average': 20},
          {'tagname': 'adult:bill', 'average': 30}
          ≫
```

11.3 Windowing cases simplified by the PartiQL grouping

Example 47. Consider again a collection of sensor readings, this time with a timestamp.

```
logs:  [{'sensor':1, 'co':0.4, 'timestamp':04:05:06},
        {'sensor':1, 'co':0.2, 'timestamp':04:05:07},
        {'sensor':1, 'co':0.5, 'timestamp':04:05:10},
        {'sensor':2, 'co':0.3}
        ]
```

We look for the "jump" readings that are more than 2x the previous reading at the same sensor. The following query solves the problem using `GROUP BY`.

```
SELECT sensor AS sensor,
       (WITH orderedReadings
             AS (SELECT v FROM oneSensorsReadings v ORDER BY v.timestamp)
        SELECT r.co, r.timestamp
        FROM orderedReadings r AT p
        WHERE r.co > 2*orderedReadings[p-1].co
        ORDER BY p
        ) AS jumpReadings
FROM logs l
GROUP BY l.sensor AS sensor GROUP AS oneSensorsReadings
```

The result is

```
≪{'sensor':1, 'jumpReadings':[{'co':0.4, 'timestamp':04:05:06}]},
 {'sensor':2, 'jumpReadings':[]}
 ≫
```

12 ORDER BY clause

SQL's **ORDER BY** orders the output data. Similarly, the PartiQL **ORDER BY** is responsible for turning its input bag into an array. In the following aspects, PartiQL extends the SQL semantics to resolve issues that are not relevant in SQL but emerge when working on Ion data.

1. SQL's **ORDER BY** clause orders its input using an expanded version of the less-than function, which we call the *order-by less-than* and denote by $<^o$. The PartiQL $<^o$ semantics (Section 12.2) also specify an order among values of heterogeneous types, including complex values.

2. The interaction of **ORDER BY** with a **UNION** (or any other set operator) of SFW queries requires attention since, unlike SQL, in PartiQL there are no binding tuples (or any tuples at all for that matter) after a **SELECT VALUE** clause. Section 12.3 elaborates on this aspect of PartiQL.

3. Unlike SQL, the input of an PartiQL query may also have order, because it is an array. The user may want to preserve the order of the input into the output. In this case, the **AT** structure in the **FROM** clause (recall, Section 5.1) can capture the input order and the **ORDER BY** can recreate it. However, this order preservation mechanism is tedious for the user. Thus, **ORDER BY** also offers an order preservation directive (Section **??**).

Sections 12.4 and 12.5 discuss SQL compatibility issues.

12.1 PartiQL ORDER BY Syntax

Similar to SQL, the PartiQL **ORDER BY** clause syntax is:

ORDER BY (e_1 [**ASC**|**DESC**]? [**NULLS FIRST**|**NULLS LAST**]?
⋮
e_m [**ASC**|**DESC**]? [**NULLS FIRST**|**NULLS LAST**]?
)
|**PRESERVE**

(Figure 3), where $e_1 \ldots e_m$ is a list of *ordering expressions*. In PartiQL a SFW query with **ORDER BY** outputs an array, whereas a SFW query without **ORDER BY** outputs a bag.

Alike SQL's **ORDER BY** clause, the **NULLS FIRST** and **NULLS LAST** keywords indicate whether **NULL** and **MISSING** values are ordered before or after all other values. Notice that in PartiQL, the **NULLS FIRST** and **NULLS LAST** refer to both **NULL** and **MISSING**.

12.2 The PartiQL order-by less-than function

The **ORDER BY** clause sorts its input using the *order-by less-than function* $<^o$, which is able to compare values of different types (unlike SQL). In particular:

1. **NULL** and **MISSING** are always first or last and compare equally according to $<^o$. In other words, $<^o$ cannot distinguish between **NULL** and **MISSING**.

2. The boolean values are coming first among the non-absent values (i.e., $b <^o x$ is always true if b is boolean and x is not a **NULL** or a **MISSING** or a boolean). **false** comes before **true**.

3. The numbers come next. The comparisons between number values do not depend on precision or specific type. Given two numbers x and y, the PartiQL $x <^o y$ behaves identical to the SQL order-by less-than function. Namely, if x and y are not the special values **'-inf'**, **'inf'** or **'nan'**, then $x <^o y$ is the same with $x<y$. The special value **'nan'** comes before **'-inf'**, which comes before all normal numeric values, which are followed by **'+inf'**.

4. Timestamp values follow and are compared by the absolute point of time irrespective of precision or local UTC offset.

5. The text types come next ordered by their lexicographical ordering by Unicode scalar irrespective of their specific type.

6. The LOB types follow and are ordered by their lexicographical ordering by octet.

7. Arrays come next, and their values compare lexicographically based on the comparison of their elements, recursively. Notice that given an array $[e_1, \ldots, e_m]$ and a longer array $[e_1, \ldots, e_m, e_{m+1}, \ldots, e_n]$ that has the same first m values, the former array comes first.

8. Tuple values follow and compare lexicographically based on the sorted attributes (as defined recursively), first by the attribute name, and secondly by the attribute values themselves.

9. Bag values come last (except, of course, when **NULLS LAST** is specified) and their values compare by first reducing them to arrays by sorting their elements and then comparing the resulting arrays.

12.3 Dependency of ORDER BY semantics on the Presence of Set Operators

Coming up...

12.4 SQL Compatibility ORDER BY clauses

For SQL-compatibility, PartiQL allows the **CURRENT** variable to be omitted from ordering expressions. Then when the **CURRENT** variable binds tuples, the ordering expressions can refer directly to the attributes of those tuples.

The complete scoping rules are as follows. When all of the following conditions are satisfied:

1. an PartiQL path expression ordering expression as appears in the **ORDER BY** of a **UNION ... ORDER BY** query, where a is an identifier and s is the potentially empty suffix of the path.

2. the expression as is evaluated in database environment ρ_0 and variables' environment ρ, which defines variables $v_1, \ldots, v_n$ and none of them is named a.

3. none of the variables $v_1, \ldots, v_n$ may bind to a tuple that has an attribute a.

then the path expression as resolves to **CURRENT**.as.

The most common and useful way to have the 3rd condition be satisfied is when the **UNION ... ORDER BY** is a top-level query and, thus, the variables environment ρ is empty.

12.5 Use of SELECT variables in ORDER BY for SQL compatibility

Recall from Section 3 that **ORDER BY** is evaluated before **SELECT**. For SQL-compatibility, given **SELECT** e_i **AS** a_i, PartiQL also supports the syntactic sugar of using a_i in lieu of e_i in the **ORDER BY** clause. Therefore, both SFW queries below are equivalent:

(1)
```
SELECT   e_i  AS  a_i
FROM     ...
ORDER BY a_i
```

(2)
```
SELECT   e_i  AS  a_i
FROM     ...
ORDER BY e_i
```

12.6 Coercion of literals for SQL compatibility

Notice that definition of `<` dismissed the SQL coercions. In SQL, given explicit literals in a query, coercions may happen.

Example 48. The query

```
SELECT * FROM foo WHERE 9 < '10'
```

is equivalent to

```
SELECT * FROM foo WHERE 9 < 10
```

because an automatic coercion of string to number will be introduced.

This aspect of SQL compatibility is introduced by rewriting. Namely, given a query with incompatible types

13 `UNION` / `INTERSECT` / `EXCEPT` clauses

Coming up...

14 PIVOT Clause Semantics

The **PIVOT** clause inputs a bag of binding tuples or an array of binding tuples. Semantically, it is similar to **SELECT VALUE** but whereas the latter creates a collection of values, **PIVOT** constructs a tuple where the each input binding is evaluated to an attribute value pair in the tuple.

The clause:

```
PIVOT v AT a
```

inputs a bag or an array of binding tuples and outputs a single tuple where each evaluation of v and a generate an attribute in the tuple.

Example 49. This example illustrates a **PIVOT** that creates a tuple from a collection of tuples.

```
PIVOT x.v AT x.a
FROM << {'a': 'first', 'v': 'john'}, {'a': 'last', 'v': 'doe'} >>
```

The result is `{'first':'john', 'last':'doe'}`.

The expression a is expected to evaluate into a string value. In strict mode, it is an error if this evaluates to a non-string value. In permissive mode, the attribute is considered **MISSING** and does not appear in the output. The expression v can be any PartiQL value, but if it is **MISSING** it will not be generated in the resulting tuple.

15 Structural Types and Type-related Query Syntax and Semantics (WIP)

The input data generally conform to a *structural type*, also often called *schema*. The SQL semantics make extensive use of the structural types in order to assign meaning to queries, which would not have a meaning in the absence of such structural types.

In the interest of SQL compatibility and user convenience, PartiQL also allows structural types to assign meaning to queries that would not have a meaning otherwise.

We will soon specify the precise rules that provide SQL compatibility, while keeping the schema optional and the query results stable with respect to schema addition.