PartiQL Tutorial

ION TEAM

2019/06/03 (00:00:33) git@62fd1f7

1 Getting Started

PartiQL provides an interactive shell, or Read Eval Print Loop (REPL), that allows users to write and evaluate PartiQL queries.

1.1 Prerequisites

PartiQL requires the Java Runtime (JVM) to be installed on your machine. You can obtain the *latest* version of the Java Runtime from either

1. OpenJDK, and OpenJDK for Windows

2. Oracle

1.2 Download the PartiQL REPL

Each release of PartiQL comes with an archive that contains the PartiQL REPL as a zip file.

1. Download the latest `partiql-cli` zip archive to your machine.
2. Expand (unzip) the archive on your machine. Expanding the archive yields the following folder structure

```
├─
 partiql-cli|
   ├─ bin|
   |    ├─ partiql|
   |    └─ partiql.bat|
   ├─ lib|
   |    └─ ...|
   ├─ README.md|
   └─ Tutorial|
   ├─       code|
   |        └─ ...|
   ├─       tutorial.html|
   └─       tutorial.pdf
```

We have used ellipsis ... to elide files/directories.

The root folder `partiql-cli` contains a `README.md` file and 3 subfolders

1. The folder `bin` contains startup scripts `partiql` for OSX (Mac) and Unix systems and `partiql.bat` for Windows systems. Execute these files to start the REPL

2. The folder `lib` contains all the necessary java libraries needed to run PartiQL.
3. The folder `Tutorial` contains the tutorial in `pdf` and `html` form. The subfolder `code` contains 3 types of files
   1. Data files with the extension `.env`. These files contains PartiQL data that we can query
   2. PartiQL query files with the extension `.sql`. These files contain the PartiQL queries used in the tutorial.
   3. Sample query output files with the extension `.out`. These files contain sample output from running the tutorial queries on the appropriate data.

1.3 Running the PartiQL REPL

1.3.1 Windows

Run (double click) on `particl.bat`. This should open a command line prompt and start the PartiQL REPL. The PartiQL REPL prompt should look like this

```
Welcome to the PartiQL REPL!
PartiQL>
```

1.3.2 OSX (Mac) and Unix

1. Open a terminal and navigate to the `partiql-cli` folder we created when we extracted `partiql-cli.zip`.
2. Run the executable `partiql` file, by typing `./partiql` and hit enter. This should start the PartiQL REPL and should look like this

```
Welcome to the PartiQL REPL!
PartiQL>
```

1.4 Testing the PartiQL REPL

Let's write a simple query to verify that our PartiQL REPL is working. At the `PartiQL>` prompt type

```
PartiQL> SELECT * FROM [1,2,3]
```

and press `ENTER` *twice*. The output should look similar to

```
PartiQL> SELECT * FROM [1,2,3]
   |
===' 
```

```
<<
  {
    '_1': 1
  },
  {
    '_1': 2
  },
  {
    '_1': 3
  }
>>
---
OK! (86 ms)
PartiQL>
```

> **INFO**
>
> An easy way to load the necessary data into the REPL while reading our Tutorial is to pass one of the .env files to the REPL on invocation, e.g. on Unix or OSX while on inside partiql-cli folder,
>
> ```
> ./bin/partiql  -e code/q1.env
> ```
>
> You can then see what is loaded in the REPL's global environment using the **special** REPL command !global_env, e.g.,
>
> ```
> Welcome to the PartiQL REPL!
> PartiQL> !global_env
>    |
> ==='
> {
>   'hr': {
>     'employees': <<
>       {
>         'id': 3,
>         'name': 'Bob Smith',
>         'title': NULL
>       },
>       {
>         'id': 4,
>         'name': 'Susan Smith',
>         'title': 'Dev Mgr'
>       },
>       {
> ```

```
        'id': 6,
        'name': 'Jane Smith',
        'title': 'Software Eng 2'
      }
    >>
  }
}
---
OK! (6 ms)
```

Congratulations! You succesfuly installed and run the PartiQL REPL. The PartiQL REPL is now waiting for more input.

To exit the PartiQL REPL press

- `Control+D` in OSX or Unix
- `Control+Z` on Windows

or close the terminal/command prompt window.

2 Introduction

PartiQL provides SQL-compatible unified query access across multiple data stores containing structured, semi-structured and nested data that is supported by SQL. PartiQL separates the syntax and semantics of a query from the underlying data source and/or data format of the data. It enables users to interact with data with[1] or without regular schema.

This tutorial aims to teach SQL users the PartiQL extensions to SQL. The tutorial is primarily driven by “how to” examples.

For the reader who is interested in the full detail and formal specification of PartiQL, we recommend the 2-tiered PartiQL formal specification: The formal specification first describes the *PartiQL core*, which is a short and concise functional programming language. Then the specification layers SQL compatibility through syntactic sugar that shows how SQL features can be translated to semantically equivalent core PartiQL expressions. These translations presented as syntactic sugar enable SQL compatibility.

[1] The implementation currently only supports data without schema. Schema support is forthcoming.

3 PartiQL Queries are SQL compatible

PartiQL is backwards compatible with SQL-92[2]. We will see what compatibility means when it is used to query data found in data formats and data stores that are not SQL.

For starters, given the table `hr.employees`

Id	name	title
3	Bob Smith	null
4	Susan Smith	Dev Mgr
6	Jane Smith	Software Eng 2

the following SQL query

```
SELECT e.id,
       e.name AS employeeName,
       e.title AS title
FROM hr.employees e
WHERE e.title = 'Dev Mgr'
```

is also a valid PartiQL query. As we know from SQL, when this query operates on the table `hr.employees` it will return the result

Id	employeeName	title
4	Susan Smith	Dev Mgr

3.1 PartiQL data model: Abstraction of many underlying data storage formats

PartiQL implementations operate not just on SQL tables but also on data that may have nesting, union types, different attributes across different tuples and many other features that we often find in today's nested and/or semi-structured formats, like JSON, Parquet, etc.

To capture this generality, PartiQL is based on a logical type system: the *PartiQL data model*. Each PartiQL implementation maps data formats, like JSON, Parquet etc., into a PartiQL data set that follows the PartiQL data model. PartiQL queries work on the PartiQL data set abstraction.

[2] SQL-92

For example, the table `hr.employees` is denoted in the PartiQL data model as this dataset

```
{
    'hr': {
        'employees': <<
            { 'id': 3, 'name': 'Bob Smith',   'title': null }, -- a tuple is denoted by { ... } in the PartiQL data model
            { 'id': 4, 'name': 'Susan Smith', 'title': 'Dev Mgr' },
            { 'id': 6, 'name': 'Jane Smith',  'title': 'Software Eng 2'}
        >>
    }
}
```

Notice that the `employees` is nested within `hr`.
The delimiters << ... >> denote that the data is an *unordered collection* (also known as *bag*), as is the case with SQL's tables. That is, there is no order between the three tuples. Single line comments start with `--` and end at the end of the line.

A very different kind of data source may lead to the same PartiQL dataset. For example, a set of JSON files that contain the following JSON objects

```
{
    ”hr” : {
        ”employees”: [
            { ”id”: 3, ”name”: ”Bob Smith”,   ”title”: null },
            { ”id”: 4, ”name”: ”Susan Smith”, ”title”: ”Dev Mgr” },
            { ”id”: 6, ”name”: ”Jane Smith”,  ”title”: ”Software Eng 2”}
        ]
    }
}
```

will likely[3] be abstracted by a PartiQL-supporting implementation into the identical PartiQL abstraction with the `hr.employees` table.

Remark: You will keep noticing the similarity of the PartiQL notation with the JSON notation. Notice also the subtle differences: In the interest of SQL compatibility, a PartiQL literal is quoted, while JSON literals are double-quoted.

Remark: You may conceptually think that a deserializer inputs JSON and outputs the PartiQL data set. But do not assume that the query processing of a PartiQL implementation will have to actually parse and

[3] The JSON value attached to `employee` is an *ordered* list. PartiQL implementations may provide their own mappings from popular data formats, e.g., CSV, TSV, JSON, Ion etc., to the PartiQL data model and/or allow clients to implements their own mappings.

abstract into PartiQL each and every bit of the underlying data storage. For example, AWS services like Redshift Spectrum and QLDB do much smarter things in order to evaluate your PartiQL queries efficiently.

Back to our PartiQL query

```
SELECT e.id,
       e.name AS employeeName,
       e.title AS title
FROM hr.employees e
WHERE e.title = 'Dev Mgr'
```

evaluates in PartiQL and returns

```
<<
  {
    'id': 4,
    'employeeName': 'Susan Smith',
    'title': 'Dev Mgr'
  }
>>
---
OK! (16 ms)
```

the result remains the same, no matter whether the `hr.employees` were the SQL table or the JSON file. All that is needed is that a catalog associates the *name* `hr.employees` with the PartiQL abstraction of the JSON data.

In the same spirit, the same PartiQL abstraction may come from a CVS file or a Parquet file, a format that has gained big traction, thanks to the efficient way in which it stores data. Again, the same query makes perfect sense, regardless of what exactly was the storage format behind `hr.employees`.

3.1.1 Learn more

- **PartiQL data sets look very much like JSON.**

  What are the differences? Indeed, PartiQL adopts the tuple/object and array notation of JSON. However, the PartiQL string literals are denoted by single quotes. Importantly, the scalar types of PartiQL are the ones of SQL and not just strings, numbers and booleans, as in JSON.

- **Do implementations need to have a catalog?**

  If queries refer to names, a catalog logically validates whether the name exists or not. However, we will also see PartiQL queries that refer to no names.

4 Querying Nested Data

SQL-92 only has tables that have tuples that contain scalar values. A key feature of many modern formats is nested data. That is, attributes whose values may be themselves tables (i.e., collections of tuples), or may be arrays of scalars, or arrays of arrays and many other combinations. We next present PartiQL's features (SQL extensions) that allow us to work with nested data.

We also include sections titled "Use Case". Such "Use Case" sections do not introduce additional features. They merely show how to combine the few novel PartiQL features with standard SQL features in order to solve a large number of problems.

4.1 Nested Collections

Let's now add the nested attribute `projects` into the data set.

```
{
  'hr': {
      'employeesNest': <<
         {
          'id': 3,
          'name': 'Bob Smith',
          'title': null,
          'projects': [ { 'name': 'AWS Redshift Spectrum querying' },
                        { 'name': 'AWS Redshift security' },
                        { 'name': 'AWS Aurora security' }
                      ]
          },
          {
              'id': 4,
              'name': 'Susan Smith',
              'title': 'Dev Mgr',
              'projects': []
          },
          {
              'id': 6,
              'name': 'Jane Smith',
              'title': 'Software Eng 2',
              'projects': [ { 'name': 'AWS Redshift security' } ]
          }
      >>
    }
}
```

Notice that the value of `'projects'` is an array. Arrays are denoted by `[ ... ]` with array elements separated by `,`. In our example the array happens to be an array of tuples. We will see that arrays may be arrays of anything, not just arrays of tuples.

4.1.1 Unnesting a Nested Collection

The query finds the names of employees who work on projects that contain the string `'security'` and outputs them along with the name of the `'security'` project. Notice that the query has just one extension over standard SQL – the `e.projects AS p` part.

```
SELECT e.name AS employeeName,
       p.name AS projectName
FROM hr.employeesNest AS e,
     e.projects AS p
WHERE p.name LIKE '%security%'
```

The output of our query is

```
<<
  {
    'employeeName': 'Bob Smith',
    'projectName': 'AWS Redshift security'
  },
  {
    'employeeName': 'Bob Smith',
    'projectName': 'AWS Aurora security'
  },
  {
    'employeeName': 'Jane Smith',
    'projectName': 'AWS Redshift security'
  }
>>
---
OK! (51 ms)
```

The extension over SQL is the `FROM` clause item `e.projects AS p`. Standard SQL would attempt to find a schema named `e` with a table `projects` and since in our example there isn't an `e.projects` table, the query would fail. In contrast, PartiQL will dereference `e.projects` to the attribute `projects` of `e`.

Once we allow this extension, the semantics are alike SQL's. The alias (also called *variable* in PartiQL) `e` gets bound to each employee, in turn. For each employee, the variable `p` gets bound to each project of the employee, in turn. Thus the query's meaning, alike SQL, is

```
foreach employee tuple e from hr.employeesNest
    foreach project tuple p from e.projects
        if p.name LIKE '%security%'
        output e.name AS employeeName, p.name AS projectName
```

Notice that our query involved variables that were ranging over nested collections (`p` in the example), along with variables that were ranging over tables (`e` in the example), as standard SQL aliases do. All variables, no matter what they range over, can be used wherever in the `FROM`, `WHERE`, `SELECT` clauses as we will see in the examples that follow.

4.1.2 Learn more

- **Can I only unnest arrays of tuples?**

  No, anything can be unnested. For example, arrays of scalars, etc.

- **Does `e.projects AS p` have to appear in the same `FROM` clause that defines `e`?**

  No. For example, see below the use cases that involve subqueries. There, the `e` and `p` are defined in separate `FROM` clauses.

- **How could I force `e.projects` to refer to the nested attribute `projects` even if there were a schema named `e` with a table `projects`?**

  Use the syntax `@e.projects`. Recall, in the absence of the `@`, in the interest of SQL compatibility, PartiQL will first attempt to dereference the `e.projects` against the catalog.

- **SQL allows me to avoid writing an explicit alias `e` when I write, say, `e.name`. Can I avoid writing the `e` in PartiQL as well?**

  SQL allows us to avoid writing aliases (variables) when the schema of the tables allows correct dereferencing. PartiQL does the same. However, recall, a schema is not necessary for a PartiQL data set. Indeed, our example has not assumed a schema. Then , in the absence of a schema, you cannot omit the aliases (variables). For example, if you write just `name` and there is no schema, PartiQL cannot tell whether you mean employee name or project name. Thus you need to explicitly write the alias (variable).

  There is one exception to this rule: If your query has a single item in its `FROM` clause, you can omit the alias (variable). Eg, you can write

  ```
  SELECT name FROM hr.employeesNest
  ```

  In this case it is apparent that `name` may only be an employee name and thus PartiQL allows you to not provide an alias (variable).

Nevertheless, for clarity we recommend that you always use aliases (variables) and this is what this tutorial does.

- **If there is a schema, can I avoid writing the alias `p`?**

  No. The `p` has to be written in order to denote the iteration over the projects.

4.1.3 Unnesting Nested Collections Using `JOIN`

In this section, we simply present an alternate way to express and think about unnesting collections.

One may think that the `FROM` clause of the example executes, in a sense, a `JOIN` between employees and projects. Except that unlike a conventional SQL join that would require an `ON` **condition**, the employees-projects join condition is implicit in the nesting of the projects data into the employee data. If it helps you to think in terms of `JOIN`, you may replace the comma with `JOIN`. That is, the following two queries are equivalent.

```
SELECT e.name AS employeeName,
       p.name AS projectName
FROM hr.employeesNest AS e,
     e.projects AS p
WHERE p.name LIKE '%security%'
```

```
SELECT e.name AS employeeName,
       p.name AS projectName
FROM hr.employeesNest AS e JOIN
    e.projects AS p
WHERE p.name LIKE '%security%'
```

4.1.4 Unnesting data with LEFT JOIN always preserves parent information

Assume that we want to write a query that returns as a bag of tuples the entire employee and project information from `hr.employeesNest`. The query result we want is this bag of tuples with attributes `id`, `employeeName`, `title` and `projectName`:

```
<<
  {
    'id': 3,
    'employeeName': 'Bob Smith',
    'title': NULL,
    'projectName': 'AWS Redshift Spectrum querying'
  },
  {
    'id': 3,
    'employeeName': 'Bob Smith',
    'title': NULL,
    'projectName': 'AWS Redshift security'
  },
  {
    'id': 3,
    'employeeName': 'Bob Smith',
    'title': NULL,
    'projectName': 'AWS Aurora security'
  },
  {
    'id': 4,
    'employeeName': 'Susan Smith',
    'title': 'Dev Mgr'
  },
  {
    'id': 6,
    'employeeName': 'Jane Smith',
    'title': 'Software Eng 2',
    'projectName': 'AWS Redshift security'
  }
>>
---
OK! (6 ms)
```

Notice that there is a `'Susan Smith'` tuple in the result, despite the fact that Susan has no project. Susan's `projectName` is **`null`**. We can obtain this result by combining employees and projects using the `LEFT JOIN` operator, as follows:

```
SELECT e.id AS id,
       e.name AS employeeName,
       e.title AS title,
       p.name AS projectName
FROM hr.employeesNest AS e LEFT JOIN e.projects AS p
```

The semantics of this query can be thought of as

```
foreach employee tuple e from hr.employeesNest
  if the e.projects is an empty collection then // this part is special about LEFT JOINs
    output e.id AS id, e.name AS employeeName, e.title AS title
    and output a null AS projectName
  else // the following part is identical to plain (inner) JOINs
    foreach project tuple p from e.projects
```

output `e.id AS id`, `e.name AS employeeName`, `e.title AS title`

and output a `null AS projectName`

4.1.5 Use Case: Checking whether a nested collection satisfies a condition

The following use cases employ the unnesting features, which we have already discussed, in new use cases. A lesson that emerges is that we can use variables (SQL aliases) that range over nested data as if they were standard SQL aliases. This realization gives us the power to solve a great number of use cases just be combining the unnesting features with features we already know from standard SQL.

In our first use case we want a query that returns the names of the employees that are involved in a project that contains the word `'security`. The solution employs SQL's "`EXISTS` (subquery)" feature, along with unnesting:

```
SELECT e.name AS employeeName
FROM hr.employeesNest AS e
WHERE EXISTS ( SELECT *
               FROM e.projects AS p
               WHERE p.name LIKE '%security%')
```

returns

```
<<
  {
    'employeeName': 'Bob Smith'
  },
  {
    'employeeName': 'Jane Smith'
  }
>>
---
OK! (14 ms)
```

In the second use case we want a query that outputs the names of the employees that have more than one security projects and we are aware of a key for employees (e.g., an attribute that is guaranteed to have a unique value for each employee). We can find the requested employees by utilizing a combination of `GROUP BY` and `HAVING`.[4] In our example, let's assume that the `id` attribute is a primary key for the employees. Then we could find the employees with more than one security project with this query:

```
SELECT e.name AS employeeName
```

[4] We could also have used the > operator with the subquery's result, but a current issue with the implementation currently prevents us from doing so.

```
FROM hr.employeesNest e,
     e.projects AS p
WHERE p.name LIKE '%security%'
GROUP BY e.id, e.name
HAVING COUNT(*) > 1
```

which returns

```
<<
  {
    'employeeName': 'Bob Smith'
  }
>>
---
OK! (28 ms)
```

4.1.6 Use Case: Subqueries that aggregate over nested collections

Next, let's find how many querying projects (that is, projects whose name contains the word "querying") each employee has.[5]

Making the same asssumption as before, that `id` is a key for employees, we can solve the problem with the query

```
SELECT e.name AS employeeName,
       COUNT(p.name) AS queryProjectsNum
FROM hr.employeesNest e LEFT JOIN e.projects AS p ON p.name LIKE '%querying%'
GROUP BY e.id, e.name
```

that returns

```
<<
  {
    'employeeName': 'Bob Smith',
    'queryProjectsNum': 1
  },
  {
    'employeeName': 'Susan Smith',
    'queryProjectsNum': 0
  },
  {
```

[5] We could also have used the > operator with the subquery's result, but a current issue with the implementation currently prevents us from doing so.

```
    'employeeName': 'Jane Smith',
    'queryProjectsNum': 0
  }
>>
---
OK! (22 ms)
```

Notice, this query’s result includes Susan Smith and Jane Smith, who have no querying projects.

4.2 Nested Tuple Values and Multi-Step Paths

A value may also be a tuple – also called object and struct in many models and formats. For example, the project value in the following tuples is always a tuple with project name and project org.

```
{
    'hr': {
        'employeesWithTuples': <<
            {
                'id': 3,
                'name': 'Bob Smith',
                'title': null,
                'project': {
                    'name': 'AWS Redshift Spectrum querying',
                    'org': 'AWS'
                }
            },
            {
                'id': 6,
                'name': 'Jane Smith',
                'title': 'Software Eng 2',
                'project': {
                    'name': 'AWS Redshift security',
                    'org': 'AWS'
                }
            }
         >>
    }
}
```

PartiQL’s multistep paths enable navigating within tuples. For example, the following query finds AWS projects and outputs the project name and employee name.

```
SELECT e.name AS employeeName,
       e.project.name AS projectName
FROM hr.employeesWithTuples e
WHERE e.project.org = 'AWS'
```

The result is

```
<<
  {
    'employeeName': 'Bob Smith',
    'projectName': 'AWS Redshift Spectrum querying'
  },
  {
    'employeeName': 'Jane Smith',
    'projectName': 'AWS Redshift security'
  }
>>
---
OK! (25 ms)
```

4.3 Unnesting Arbitrary Forms of Nested Collections

The previous examples have shown nested attributes that were arrays of tuples. It need not be the case that the nested attributes are collections of tuples. They may just as well be arrays of scalars, arrays of arrays, and more. In general any combination of data that one can create by composing scalars, tuples and arrays. You need not learn a different set of query language features for each case. The unnesting features, which we have already seen, are sufficient.

4.3.1 Use Case: Unnesting Arrays of Scalars

The list of projects associated with each employee in `hr.employeesNest` could have been simply a list of project name strings. Replacing the nested tuples with plain strings gives us

```
{
    'hr': {
        'employeesNestScalars': <<
            {
                'id': 3,
                'name': 'Bob Smith',
                'title': null,
                'projects': [
                    'AWS Redshift Spectrum querying',
                    'AWS Redshift security',
                    'AWS Aurora security'
                ]
            },
            {
                'id': 4,
                'name': 'Susan Smith',
                'title': 'Dev Mgr',
                'projects': []
            },
            {
                'id': 6,
                'name': 'Jane Smith',
                'title': 'Software Eng 2',
                'projects': [ 'AWS Redshift security' ]
            }
        >>
    }
}
```

Let us repeat the previous use cases on the revised employee data.

The following query finds the names of employees who work on projects that contain the string `'security'` and outputs them along with the name of the "security" project.

```
SELECT e.name AS employeeName,
       p AS projectName
FROM hr.employeesNestScalars AS e,
     e.projects AS p
WHERE p LIKE '%security%'
```

The preceding query returns

```
<<
  {
    'employeeName': 'Bob Smith',
    'projectName': 'AWS Redshift security'
  },
  {
    'employeeName': 'Bob Smith',
    'projectName': 'AWS Aurora security'
  },
  {
    'employeeName': 'Jane Smith',
    'projectName': 'AWS Redshift security'
  }
>>
---
OK! (28 ms)
```

The variable p ranges (again) over the content of e.projects. In this case, since e.projects has strings (as opposed to tuples), the variable p binds each time to a project name string. Thus, this query can be thought of as executing the following snippet.

```
foreach employee tuple e from hr.employeesNestScalars
  foreach project p from e.projects
    if the string p matches '%security%'
    output e.name AS employeeName and the string p AS projectName
```

4.3.2 Use Case: Unnesting Arrays of Arrays

Arrays may also contain arrays, directly, without intervening tuples, as in the matrices data set.

```
{
    'matrices': <<
        {
            'id': 3,
            'matrix': [
                [2, 4, 6],
                [1, 3, 5, 7],
                [9, 0]
            ]
        },
        {
            'id': 4,
            'matrix': [
                [5, 8],
                [ ]
            ]

        }
    >>
}
```

The following query finds every even number and outputs the even number and the `id` of the tuple where it was found.

```
SELECT t.id AS id,
       x AS even
FROM matrices AS t,
     t.matrix AS y,
     y AS x
WHERE x % 2 = 0
```

The preceding query returns

```
<<
  {
    'id': 3,
    'even': 2
  },
  {
    'id': 3,
    'even': 4
  },
  {
    'id': 3,
    'even': 6
  },
  {
    'id': 3,
    'even': 0
  },
  {
    'id': 4,
    'even': 8
  }
>>
---
OK! (59 ms)
```

Informally the query's evaluation can be thought of as

```
foreach tuple t from matrices
  foreach array y from t.matrix
    foreach number x from y
      if x is even then
```

output `t.id AS id` and `x AS even`

5 Literals

Literals of the PartiQL query language correspond to the types in the PartiQL data model:

- scalars, including **`null`** which follow the SQL syntax when applicable. For example:
  - `5`
  - `'foo'`
- tuples, denoted by `{...}` with tuple elements separated by `,` (also known as structs and/or objects in many formats and other data models)
  - `{ 'id': 3, 'arr': [1, 2] }`
- arrays, denoted by `[...]` with array elements separated by `,`
  - `[ 1, 'foo']`
- bags, denoted by `<< ... >>` with bag elements separated by a `,`
  - `<< 1, 'foo'>>`

Notice that in the spirit of the PartiQL data model, literals compose freely and any kind of literal may appear within any tuple, array and bag literal, eg.,

```
{
    'id': 3,
    'matrix': [
        [2, 4, 6],
        'NA'
    ]
}
```

6 Querying Heterogeneous and Schemaless Data

Many formats do not require a schema that describes the data – that is *schemaless data*. In such cases it is possible to have various “heterogeneities” in the data:

- One tuple may have an attribute `x` while another tuple may not have this attribute
- In one tuple of a collection an attribute `x` may be of type, e.g., string, while in another tuple of the same collection the same attribute `x` may be of a different type – e.g, array.

- The elements of a collection (be it a bag or array) can be heterogeneous (not have the same type). For example, the first element may be a string, the second one may be an integer and the third one an array.
- Generally, any composition is possible as we can bundle heterogeneous elements in arrays and bags.

Heterogeneities are not particular to schemaless. Schemas may allow for heterogeneity in the types of the data. For example, one of the Hive data types is the union type,[6] which allows a value to belong to any one of a list of types. For example, in the following schema the `projects` attribute may be either a string or an array of strings

```
CREATE TABLE employeesMixed(
        id: INT,
        name: STRING,
        title: STRING,
        projects: UNIONTYPE<STRING, ARRAY<STRING>>
);
```

A collection of PartiQL tuples that follows this schema could be

```
{
    'hr': {
        'employeesMixed1': <<
            {
                'id': 3,
                    'name': 'Bob Smith',
                    'title': null,
                    'projects': [
                        'AWS Redshift Spectrum querying',
                    'AWS Redshift security',
                    'AWS Aurora security'
                    ]
            },
            {
                'id': 4,
                'name': 'Susan Smith',
                'title': 'Dev Mgr',
                'projects': []
            },
            {
                'id': 6,
```

[6] Hive Union Type

```
                'name': 'Jane Smith',
                'title': 'Software Eng 2',
                'projects': 'AWS Redshift security'
            }
        >>
    }
}
```

Thus we see that data may have heterogeneities – regardless of whether they are described by a schema or not. PartiQL tackles heterogeneous data, in ways that we will see in the next use cases and feature presentations.

6.1 Tuples with Missing Attributes

Let's go back to the `hr.employees` table (that is, bag of tuples). Bob Smith has no title and, as is typical in SQL, the lack of title is modeled with the **`null`** value.

```
{
    'hr': {
        'employees': <<
            { 'id': 3, 'name': 'Bob Smith',   'title': null }
            { 'id': 4, 'name': 'Susan Smith', 'title': 'Dev Mgr' }
            { 'id': 6, 'name': 'Jane Smith',  'title': 'Software Eng 2'}
        >>
    }
}
```

Nowadays, many semi-structured formats allow users to represent "missing" information in two ways.

1. The first way is by use of **`null`**.
2. The second kind is the plain absence of the attribute from the tuple.

That is, we can represent the fact that Bob Smith has no title by simply having no `title` attribute in the `'Bob Smith'` tuple:

```
{
    'hr': {
        'employeesWithMissing': <<
            { 'id': 3, 'name': 'Bob Smith' }, -- no title in this tuple
            { 'id': 4, 'name': 'Susan Smith', 'title': 'Dev Mgr' },
            { 'id': 6, 'name': 'Jane Smith', 'title': 'Software Eng 2'}
        >>
    }
}
```

PartiQL does not argue about when to use `null`s and when to use “missing”. Myriads of datasets already use one of the two or both. However, PartiQL enables queries to distinguish when they access a `null` Vs when they access a missing attribute. PartiQL also enables queries to create results that have both `null`s and missing attributes. Indeed, it makes it very easy to propagate source data `null`s as query result `null`s and source data missing attributes into result missing attributes.

6.2 Accessing and Processing Missing Attributes: The MISSING Value

Consider again this PartiQL query, which happens to also be an SQL query.

```
SELECT e.id,
       e.name AS employeeName,
       e.title AS title
FROM hr.employeesWithMissing AS e
WHERE e.title = 'Dev Mgr'
```

What will happen when the query goes over the Bob Smith tuple, which has no `title`?

The first step to answering this question is understanding the result of the path `e.title` when the alias (variable) `e` binds to the tuple `{ 'id': 3, 'name': 'Bob Smith'}`. In more basic terms, what is the result of the expression `{ 'id': 3, 'name': 'Bob Smith'}.path` ? PartiQL says that it is the special value `MISSING`. `MISSING` behaves very similar to `null`.

6.2.1 Evaluating Functions and Conditions with MISSING

If a function (including infix functions like =) inputs a `MISSING` the function’s result is also `MISSING`. In the case of the example, this means that the `WHERE` clause `e.title='Dev Mgr'` will evaluate to `MISSING` when `e` binds to `{ 'id': 3, 'name': 'Bob Smith'}` and, as usual in SQL, the `WHERE` clause fails when it does not evaluate to `true`. Thus the output will be

```
<<
  {
    'id': 4,
    'employeeName': 'Susan Smith',
    'title': 'Dev Mgr'
  }
>>
---
OK! (17 ms)
```

6.2.2 Propagating MISSING in Result Tuples

What would happen if a missing attribute or, more generally, an expression returning MISSING appears in the SELECT?

```
SELECT e.id,
       e.name AS employeeName,
       e.title AS outputTitle
FROM hr.employeesWithMissing AS e
```

The query will output one tuple for each employee. When it outputs the Bob Smith tuple, the e.title will evaluate to MISSING and then the output tuple will not even have an outputTitle attribute.

```
<<
  {
    'id': 3,
    'employeeName': 'Bob Smith'
  },
  {
    'id': 4,
    'employeeName': 'Susan Smith',
    'outputTitle': 'Dev Mgr'
  },
  {
    'id': 6,
    'employeeName': 'Jane Smith',
    'outputTitle': 'Software Eng 2'
  }
>>
---
OK! (10 ms)
```

The same treatment of MISSING would happen if, say, we had this query that converts titles to capital letters:

```
SELECT e.id,
       e.name AS employeeName,
       UPPER(e.title) AS outputTitle
FROM hr.employeesWithMissing AS e
```

Again, the e.title will evaluate to MISSING for 'Bob Smith', the UPPER(e.title) is then UPPER(MISSING) and also evaluates to MISSING. Thus the result will be

```
<<
  {
    'id': 3,
    'employeeName': 'Bob Smith',
    'outputTitle': NULL
  },
  {
    'id': 4,
    'employeeName': 'Susan Smith',
    'outputTitle': 'DEV MGR'
  },
  {
    'id': 6,
    'employeeName': 'Jane Smith',
    'outputTitle': 'SOFTWARE ENG 2'
  }
>>
---
OK! (20 ms)
```

6.3 Variables can range over Data with Different Types

A PartiQL variable (called *alias* in SQL) can bind to data of different types during a query's evaluation. This is unlike SQL where the variables always bind to tuples. It is even different from what happened in Use Case: Unnesting Arrays of Scalars and what happened in Use Case: Unnesting Arrays of Arrays.

In the first use case, the PartiQL variable `p` happened to always bind to a string (given the particular sample data of the example). In the second use case, the PartiQL variable `y` was always bound to an array (again, given the particular sample data of the example).

To make the case for variables that bind to different types, consider the following twist in the `employeesNest` data set. Some of the elements of the `projects` array are plain strings and some are tuples. Even the employee tuples do not always have the same attributes.

```
{
    'hr': {
        'employeesMixed2': <<
            {
                'id': 3,
                'name': 'Bob Smith',
                'title': null,
                'projects': [
                    { 'name': 'AWS Redshift Spectrum querying' },
                    'AWS Redshift security',
                    { 'name': 'AWS Aurora security' }
                ]
            },
            {
                'id': 4,
                'name': 'Susan Smith',
                'title': 'Dev Mgr',
                'projects': []
            },
            {
                'id': 6,
                'name': 'Jane Smith',
                'projects': [ 'AWS Redshift security']
            }
        >>
    }
}
```

This query on `hr.employeesMixed2` produces employee name – employee project pairs.

```
SELECT e.name AS employeeName,
       CASE WHEN (p IS TUPLE) THEN p.name
       ELSE p END AS projectName
FROM hr.employeesMixed2 AS e,
     e.projects AS p
```

Notice the sub-expression `(p IS TUPLE)`. The `IS` operator can be used to check a value against it's type at evaluation time. Notice also that the variable `p` binds to different types.

In general, the `FROM` clause of a query binds its variables (aliases) to data. The variables need not bind to data that have the same types. Each binding is fed to the `SELECT` clause, which evaluates its expressions.

This table shows each variables' binding produced by the `FROM` clause and the corresponding tuple output by the `SELECT` clause.

Variable e	Variable p	Result tuple

{ 'id': 3,
'name': 'Bob Smith',
'title': **null**,
'projects': [{
'name': 'AWS Redshift Spectrum querying' },
'AWS Redshift security',
{ 'name': 'AWS Aurora security' }
 ]
}
	{ 'name': 'AWS Redshift Spectrum querying' }	{
		'employeeName': 'Bob Smith',
		'projectName': 'AWS Redshift Spectrum querying'
		}

Variable `e`	Variable `p`	Result tuple

`{ 'id': 3,`
 	
`'name': 'Bob Smith',`
 	
`'title': null,`
 	
`'projects':  [ {`
`'name': 'AWS Redshift`
`Spectrum querying' },`
 	
`'AWS Redshift`
`security',`
 	
`{ 'name': 'AWS Aurora`
`security' }`
 	
` ]`
 	
`}`
	`'AWS Redshift`
	`security'`
		`{`
		
		`'employeeName': 'Bob`
		`Smith',`
		
		`'projectName': 'AWS`
		`Redshift security'`
		
		`}`

Variable `e`	Variable `p`	Result tuple

`{ 'id': 3,`
 	
`'name': 'Bob Smith',`
 	
`'title': null,`
 	
`'projects': \[ {`
`'name': 'AWS Redshift`
`Spectrum querying' },`
 	
`'AWS Redshift`
`security',`
 	
`{ 'name': 'AWS Aurora`
`security' }`
 	
`\]`
 	
`}`
	`{ 'name': 'AWS Aurora`
	`security' }`
		`{`
		
		`'employeeName': 'Bob`
		`Smith',`
		
		`'projectName': 'AWS`
		`Aurora security'`
		
		`}`

`{ 'id': 6,`
 	
`'name': 'Jane Smith',`
 	
`'projects': \[ 'AWS`
`Redshift security' \]`
 	
`}`
	`  'AWS Redshift`
	`security'`
		`{`
		
		`  'employeeName':`
		`'Jane Smith',`
		
		`  'projectName': 'AWS`
		`Redshift security'`
		
		`}`

7 Accessing Array Elements by Order

SQL allows us to order the output of a query using the `ORDER BY` clause. However, the SQL data model does not recognize order in the input data. In contrast, many of the new data formats feature arrays; the array’s elements have an order. We may want to find an array element according to its order, or, we may want to find the positions of certain elements in their arrays.

7.1 `<Array> [<number>]`

Let’s consider again the dataset `hr.employeesNest`.

```
{
  'hr': {
      'employeesNest': <<
         {
          'id': 3,
          'name': 'Bob Smith',
          'title': null,
          'projects': [ { 'name': 'AWS Redshift Spectrum querying' },
                        { 'name': 'AWS Redshift security' },
                        { 'name': 'AWS Aurora security' }
                      ]
          },
          {
              'id': 4,
              'name': 'Susan Smith',
              'title': 'Dev Mgr',
              'projects': []
          },
          {
              'id': 6,
              'name': 'Jane Smith',
              'title': 'Software Eng 2',
              'projects': [ { 'name': 'AWS Redshift security' } ]
          }
      >>
    }
}
```

The `projects` attribute is an array of tuples; that is, each tuple has an ordinal associated with it. The following query returns each employee name, along with the first project of the employee.

```
SELECT e.name AS employeeName,
       e.projects[0].name AS firstProjectName
FROM hr.employeesNest AS e
```

The query returns

```
<<
  {
    'employeeName': 'Bob Smith',
    'firstProjectName': 'AWS Redshift Spectrum querying'
  },
  {
    'employeeName': 'Susan Smith'
  },
  {
    'employeeName': 'Jane Smith',
    'firstProjectName': 'AWS Redshift security'
  }
>>
---
OK! (51 ms)
```

7.2 Multistep Paths

Technically, the structure `[<number>]` is a kind of path step. For example, notice the 4-step path `e.projects[0].name`. When `e` is bound to the first tuple of `hr.employeesNest`, then the path `e.projects` results into the array

```
[
    { 'name': 'AWS Redshift Spectrum querying' },
    { 'name': 'AWS Redshift security' },
    { 'name': 'AWS Aurora security' }
]
```

Consequently applying the `[0]` step on `e.projects` (that is, evaluating `e.projects[0]`) leads to `{'name': 'AWS Redshift Spectrum querying'}`. Finally, evaluating the `.name` step on `e.projects[0]` (that is, evaluating `e.projects[0].name`) leads to `'AWS Redshift Spectrum querying'`.

7.3 Finding the Order of Each Element in an Array

Let's assume that the order of each employee's projects in the `projects` attribute of `hr.employeesNest` matters. The first project is the employee's highest priority project, followed by the second and so on. The following query finds the names of each employee involved in a security project, the security project and its order in the `projects` array.

```
SELECT e.name AS employeeName,
       p.name AS projectName,
       o AS projectPriority
FROM hr.employeesNest AS e,
     e.projects AS p AT o
WHERE p.name LIKE '%security%'
```

Notice the new feature: `AT o`. While `p` ranges over the elements of the array `e.projects`, the variable `o` takes as value the ordinal number of the element in the array. The query returns

```
<<
  {
    'employeeName': 'Bob Smith',
    'projectName': 'AWS Redshift security',
    'projectPriority': 1
  },
  {
    'employeeName': 'Bob Smith',
    'projectName': 'AWS Aurora security',
    'projectPriority': 2
  },
  {
    'employeeName': 'Jane Smith',
    'projectName': 'AWS Redshift security',
    'projectPriority': 0
  }
>>
---
OK! (12 ms)
```

8 Pivoting & Unpivoting

Many queries need to range over and collect the attribute name/value pairs of tuples or the key/value pairs of maps.

8.1 Unpivoting Tuples

Consider this dataset that provides the closing prices of multiple ticker symbols.

```
{
    'closingPrices': <<
        { 'date': '4/1/2019', 'amzn': 1900, 'goog': 1120, 'fb': 180 },
        { 'date': '4/2/2019', 'amzn': 1902, 'goog': 1119, 'fb': 183 }
    >>
}
```

The following query unpivots the stock ticker/price pairs.

```
SELECT c.”date” AS ”date”,
       sym AS ”symbol”,
       price AS price
FROM closingPrices AS c,
     UNPIVOT c AS price AT sym
WHERE NOT sym = 'date'
```

Notice the use of ” in this query. The double quotes allow us to disambiquate from `date` the keyword and `”date”` the identifier. Also double quote specify case sensitivity for attribute lookups.

The query returns

```
<<
  {
    'date': '4/1/2019',
    'symbol': 'amzn',
    'price': 1900
  },
  {
    'date': '4/1/2019',
    'symbol': 'goog',
    'price': 1120
  },
  {
    'date': '4/1/2019',
    'symbol': 'fb',
    'price': 180
  },
  {
    'date': '4/2/2019',
    'symbol': 'amzn',
    'price': 1902
  },
  {
    'date': '4/2/2019',
    'symbol': 'goog',
    'price': 1119
  },
  {
    'date': '4/2/2019',
    'symbol': 'fb',
    'price': 183
  }
>>
---
OK! (18 ms)
```

Unpivoting tuples enables the use of attribute names as if they were data. For example, it becomes easy to compute the average price for each symbol as

```
SELECT sym AS ”symbol”,
       AVG(price) AS avgPrice
FROM closingPrices c,
     UNPIVOT c AS price AT sym
WHERE NOT sym = 'date'
GROUP BY sym
```

which returns

```
<<
  {
    'symbol': 'amzn',
    'avgPrice': 1901
  },
  {
    'symbol': 'fb',
    'avgPrice': 181.5
  },
  {
    'symbol': 'goog',
    'avgPrice': 1119.5
  }
>>
---
OK! (31 ms)
```

8.2 Pivoting into Tuples

Pivoting turns a collection into a tuple. For example, consider the collection

```
{
    'todaysStockPrices': <<
        { 'symbol': 'amzn', 'price': 1900},
        { 'symbol': 'goog', 'price': 1120},
        { 'symbol': 'fb', 'price': 180 }
    >>
}
```

Then the following PIVOT query

```
PIVOT sp.price AT sp.”symbol”
FROM todaysStockPrices sp
```

produces the tuple

```
{
  'amzn': 1900,
  'goog': 1120,
  'fb': 180
}
---
OK! (43 ms)
```

Notice that the PIVOT query looks like a SELECT-FROM-WHERE-... query except that instead of a SELECT clause it has a PIVOT <value expression> AT <attribute expression>. Note also that the PIVOT query does not return a singleton collection of tuples: Rather it literally returns a tuple value.

8.3 Use Case: Pivoting Subqueries

(This example also uses the grouping features of PartiQL, Creating Nested Results with GROUP BY … GROUP AS.)

Let us generalize the previous case of pivoting. We have a table of stock prices

```
{
    'stockPrices':<<
        { 'date': '4/1/2019', 'symbol': 'amzn', 'price': 1900},
        { 'date': '4/1/2019', 'symbol': 'goog', 'price': 1120},
        { 'date': '4/1/2019', 'symbol': 'fb',   'price': 180 },
        { 'date': '4/2/2019', 'symbol': 'amzn', 'price': 1902},
        { 'date': '4/2/2019', 'symbol': 'goog', 'price': 1119},
        { 'date': '4/2/2019', 'symbol': 'fb',   'price': 183 }
    >>
}
```

and we want to pivot it into a collection of tuples, where each tuple has all the `symbol:price` pairs for a date, as follows

```
<<
{
    'date': date(4/1/2019),
    'prices': {'amzn': 1900, 'goog': 1120, 'fb': 180}
},
{
    'date': date(4/2/2019),
    'prices': {'amzn': 1902, 'goog': 1119, 'fb': 183}
}
>>
```

The following query first creates one group datesPrices for each date. Then the `PIVOT` subquery pivots the group into the tuple prices.

```
SELECT sp.”date” AS ”date”,
       (PIVOT dp.sp.price AT dp.sp.”symbol”
        FROM datesPrices as dp ) AS prices
FROM StockPrices AS sp GROUP BY sp.”date” GROUP AS datesPrices
```

For example, the `datesPrices` collection, returned from `GROUP AS` for `sp.date = date(4/1/2019)` is

```
    'datesPrices': <<
      {
        'sp': {
          'date': '4/1/2019',
          'symbol': 'amzn',
          'price': 1900
        }
      },
      {
        'sp': {
          'date': '4/1/2019',
          'symbol': 'goog',
          'price': 1120
        }
      },
      {
        'sp': {
          'date': '4/1/2019',
          'symbol': 'fb',
          'price': 180
        }
      }
    >>
```

9 Creating Nested and Non-SQL Results

PartiQL allows queries that create nested results as well as queries that create heterogeneous results.

9.1 Creating Nested Results with `SELECT VALUE` Queries

Let’s consider again the dataset `hr.employeesNestScalars`:

```
{
    'hr': {
        'employeesNestScalars': <<
            {
                'id': 3,
                'name': 'Bob Smith',
                'title': null,
                'projects': [
                    'AWS Redshift Spectrum querying',
                    'AWS Redshift security',
                    'AWS Aurora security'
                ]
            },
            {
                'id': 4,
                'name': 'Susan Smith',
                'title': 'Dev Mgr',
                'projects': []
            },
            {
                'id': 6,
                'name': 'Jane Smith',
                'title': 'Software Eng 2',
                'projects': [ 'AWS Redshift security' ]
            }
        >>
    }
}
```

The following query outputs each tuple of `hr.employeesNestScalars`, except that instead of all projects each tuple has only the security projects of the employee. The important new feature here is the `SELECT VALUE <expression>`.

```
SELECT e.id AS id,
       e.name AS name,
       e.title AS title,
       ( SELECT VALUE p
         FROM e.projects AS p
         WHERE p LIKE '%security%'
       ) AS securityProjects
FROM hr.employeesNestScalars AS e
```

The result is

```
<<
  {
    'id': 3,
    'name': 'Bob Smith',
    'title': NULL,
    'securityProjects': <<
      'AWS Redshift security',
      'AWS Aurora security'
    >>
  },
  {
    'id': 4,
    'name': 'Susan Smith',
    'title': 'Dev Mgr',
    'securityProjects': <<>>
  },
  {
    'id': 6,
    'name': 'Jane Smith',
    'title': 'Software Eng 2',
    'securityProjects': <<
      'AWS Redshift security'
    >>
  }
```

```
>>
---
OK! (35 ms)
```

A `SELECT VALUE <expression>` query (or subquery, as in this example) returns a collection of whatever the `<expression>` evaluates to.

Notice the difference from SQL's `SELECT`, which always produces tuples. If a SQL `SELECT` appears as a subquery, then the context of the subquery designates whether the subquery's result should be coerced into a scalar (e.g., when `5 = <subquery>`), coerced into a collection of scalars (e.g., when `5 IN <subquery>`), etc. None of this applies to `SELECT VALUE`, which produces a collection and this collection is not coerced.

9.2 Creating Nested Results with GROUP BY ... GROUP AS

Another pattern of creating nested results in PartiQL is via the `GROUP AS` extension to SQL's `GROUP BY`. This pattern is more efficient and more intuitive than the use of nested `SELECT VALUE` queries when the required nesting is not following the nesting of the input. (The example in Creating Nested Results with `SELECT VALUE` Queries is one where the nesting in the output follows the nesting of the input and, thus, an intuitive solution does not involve `GROUP BY`.)

The following query outputs each security project found in `hr.employeesNestScalars` along with the list of employee names that work on the project.

```
SELECT p AS projectName,
       ( SELECT VALUE v.e.name
         FROM perProjectGroup AS v ) AS employees
FROM hr.employeesNestScalars AS e JOIN e.projects AS p ON p LIKE '%security%'
GROUP BY p GROUP AS perProjectGroup
```

The result is

```
<<
  {
    'projectName': 'AWS Aurora security',
    'employees': <<
      'Bob Smith'
    >>
  },
  {
    'projectName': 'AWS Redshift security',
    'employees': <<
      'Bob Smith',
      'Jane Smith'
    >>
  }
>>
---
OK! (24 ms)
```

The `GROUP AS` generalizes SQL's `GROUP BY` by making the formulated groups available in their entirety to the query's `SELECT` and `HAVING` clauses. Contrast with SQL's `GROUP BY`, where the `SELECT` and `HAVING` clauses can have aggregate functions over grouped columns but they cannot get access to the individual values of the grouped columns.

To better understand the workings of `GROUP BY ... GROUP AS` it is best to think of PartiQL queries as a pipeline of clauses, starting with the `FROM`, continuing with the `GROUP BY` and finishing with the `SELECT`. Each clause is a function that inputs data and outputs data. In that sense, the `GROUP BY ... GROUP AS` is a function that inputs the result of the `FROM` and outputs its result to the `SELECT`.

The following query (conceptually) produces the output of the `FROM` clause.

```
SELECT e AS e, p AS p
FROM hr.employeesNestScalars AS e JOIN e.projects AS p ON p LIKE '%security%'
```

We see that the `FROM` delivers the collection of tuples consisting of an employee `e` and a project `p` that were output by the `FROM` clause, i.e., the `LEFT JOIN`. This is alike SQL's `FROM` semantics.

Variable e	Variable p
{ 'id': 3, 'name': 'Bob Smith', 'title': null, 'projects': ['AWS Redshift Spectrum querying', 'AWS Redshift security', 'AWS Aurora security'] }	'AWS Redshift security'
{ 'id': 3, 'name': 'Bob Smith', 'title': null, 'projects': ['AWS Redshift Spectrum querying', 'AWS Redshift security', 'AWS Aurora security'] }	'AWS Aurora security'

Variable `e`	Variable `p`

`{ 'id': 6,`
`'name': 'Jane Smith',`
`'title': 'Software Eng 2',`
`'projects':  [ 'AWS Redshift`
`security'  ]`
`}`
	`'AWS Redshift security'`

Then the `GROUP BY ... GROUP AS ...` can be thought of as outputting a table that has one column for each group-by expression (i.e., each security project `p`) and a last column `perProjectGroup` whose value (conceptually) is the collection of employee/project `e`/`p` tuples that correspond to the group-by expression `p`. Thus the `GROUP BY ... GROUP AS ...` output is the table

`p`	`perProjectGroup`

`'AWS Redshift security'`	`<<`
	`{ e: { 'id': 3, 'name': 'Bob`
	`Smith', ... }, p: 'AWS Redshift`
	`security' },`
	`{ e: { 'id': 6, 'name': 'Jane`
	`Smith', ... }, p: 'AWS Redshift`
	`security' }`
	`>>`

p	perProjectGroup
'AWS Aurora security'	<< { e: { 'id': 3, 'name': 'Bob Smith', ...}, p: 'AWS Aurora security' }, >>

Finally the `SELECT` clause inputs the above and outputs the query result.