PartiQL User Guide

Ion Team

1 Preface

The user's guide provides an explanation of the features implemented by PartiQL's kotlin implementation.

This document is an early draft, contributions welcome!

1.1 What is PartiQL

PartiQL is an implementation of SQL++ based upon Ion's type system. PartiQL is based on SQL92 and provides support for working with schemaless hierarchical data.

1.2 Conventions

TBD

1.3 Further Reading

- SQL++
- Ion's

1.4 Bug Reports

We welcome you to use the GitHub issue tracker to report bugs or suggest features.

When filing an issue, please check existing open, or recently closed, issues to make sure somebody else hasn't already reported the issue. Please try to include as much information as you can. Details like these are incredibly useful:

- A reproducible test case or series of steps
- The version of our code being used
- Any modifications you've made relevant to the bug
- Anything unusual about your environment or deployment

1.5 Contribute

See our contribute guide.

2 FAQ

TBD

3 Getting Started

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

3.1 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, or OpenJDK for Windows

2. Oracle

Follow the instructions on how to set `JAVA_HOME` to the path where your Java Runtime is installed.

3.2 Download the PartiQL REPL

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

1. Download. You may have to click on `Assets` to see the zip and tgz archives. the latest `partiql-cli`[1] 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
```

[1]The file will append PartiQL's release version to the archive, i.e., `partiql-cli-0.1.0.zip`.


```
    │       └─ ...
    ├─      tutorial.html
    └─      tutorial.pdf
```

where ... represents elided files/directories.

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

1. The folder `bin` contains startup scripts `partiql` for macOS 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 `.output`. These files contain sample output from running the tutorial queries on the appropriate data.

3.3 Running the PartiQL REPL

3.3.1 Windows

Run (double click on) `particl.bat`. This should open a command-line prompt and start the PartiQL REPL which displays:

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

3.3.2 macOS (Mac) and Unix

1. Open a terminal and navigate to the `partiql-cli`[2] folder.
2. Start the REPL by typing `./bin/partiql` and pressing ENTER, which displays:

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

[2]The folder name will have the PartiQL version as a suffix, i.e., `partiql-cli-0.1.0`.

3.4 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!
PartiQL>
```

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

To exit the PartiQL REPL, press:

- `Control+D` in macOS or Unix
- `Control+C` on Windows

or close the terminal/command prompt window.

3.5 Loading data from a file

An easy way to load the necessary data into the REPL is use the `-e` switch when starting the REPL and provide the name of a file that contains your data.

```
./bin/partiql  -e Tutorial/code/q1.env
```

You can then see what is loaded in the REPL's global environment using the **special** REPL command `!global_env`, i.e.,

```
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!
```

4 PartiQL CLI

```
PartiQL CLI
Command line interface for executing PartiQL queries. Can be run in an interactive (REPL)
    mode or non-interactive.

Examples:
To run in REPL mode simply execute the executable without any arguments:
     partiql
```

```
In non-interactive mode we use Ion as the format for input data which is bound to a global
    variable
named "input_data", in the example below /logs/log.ion is bound to "input_data":
     partiql --query="SELECT * FROM input_data" --input=/logs/log.ion

The cli can output using PartiQL syntax or Ion using the --output-format option, e.g. to
    output binary ion:
     partiql --query="SELECT * FROM input_data" --output-format=ION_BINARY --input=/logs/
         log.ion

To pipe input data in via stdin:
     cat /logs/log.ion | partiql --query="SELECT * FROM input_data" --format=ION_BINARY >
         output.10n

Option                                Description
------                                -----------
-e, --environment <File>              initial global environment (optional)
-h, --help                            prints this help
-i, --input <File>                    input file, requires the query option (default: stdin
    )
-o, --output <File>                   output file, requires the query option (default:
    stdout)
--of, --output-format <OutputFormat:  output format, requires the query option (default:
    PARTIQL)
  (ION_TEXT|ION_BINARY|PARTIQL)>
-q, --query <String>                  PartiQL query, triggers non interactive mode
```

5 Building the CLI

The CLI is built during the main Gradle build. To build it separately execute:

```
./gradlew :cli:build
```

After building, distributable jars are located in the `cli/build/distributions` directory (relative to the project root).

Be sure to include the correct relative path to `gradlew` if you are not in the project root.

6 Using the CLI

The following command will build any dependencies before starting the CLI.


```
./gradlew :cli:run -q --args="<command line arguments>"
```

7 REPL

To start an interactive read, eval, print loop (REPL) execute:

```
rlwrap ./gradlew :cli:run --console=plain
```

rlwrap provides command history support. It allows the use of the up and down arrow keys to cycle through recently executed commands and remembers commands entered into previous sessions. `rlwrap` is available as an optional package in all major Linux distributions and in Homebrew on MacOS. `rlwrap` is not required but is highly recommended.

You will see a prompt that looks as follows:

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

At this point you can type in SQL and press enter *twice* to execute it:

```
PartiQL> SELECT id FROM `[{id: 5, name:"bill"}, {id: 6, name:"bob"}]` WHERE name = 'bob'
    |
==='
<<
  {
    'id': 6
  }
>>
---
OK!
```

The result of previous expression is stored in the variable named `_`, so you can then run subsequent expressions based on the last one.

```
PartiQL> SELECT id + 4 AS name FROM _
    |
==='
<<
  {
    'name': 10
  }
```

```
>>
---
OK!
```

Press control-D to exit the REPL.

7.1 Advanced REPL Features

To view the AST of an SQL statement, type one and press enter only *once*, then type `!!` and press enter:

```
PartiQL> 1 + 1
   | !!
==='

(
  ast
  (
    version
    1
  )
  (
    root
    (
      +
      (
        lit
        1
      )
      (
        lit
        1
      )
    )
  )
)
---
OK!
```

To view the AST with metadata information of an SQL statement, type one and press enter only *once*, then type `!?` and press enter:

```
PartiQL> 1 + 1
   | !?
```

```
===' 

(
  ast
  (
    version
    1
  )
  (
    root
    (
      term
      (
        exp
        (
          +
          (
            term
            (
              exp
              (
                lit
                1
              )
            )
            (
              meta
              (
                $source_location
                (
                  {
                    line_num:1,
                    char_offset:1
                  }
                )
              )
            )
          )
          (
            term
            (
              exp
              (
```

```
                lit
                1
              )
            )
            (
              meta
              (
                $source_location
                (
                  {
                    line_num:1,
                    char_offset:5
                  }
                )
              )
            )
          )
        )
      )
      (
        meta
        (
          $source_location
          (
            {
              line_num:1,
              char_offset:3
            }
          )
        )
      )
    )
  )
)
---
OK!
```

7.2 Initial Environment

The initial environment for the REPL can be setup with a configuration file, which should be an PartiQL file with a single `struct` containing the initial *global environment*.

For example a file named `config.sql`, containing the following:

```
{
  'animals':[
    {'name': 'Kumo', 'type': 'dog'},
    {'name': 'Mochi', 'type': 'dog'},
    {'name': 'Lilikoi', 'type': 'unicorn'}
  ],
  'types':[
    {'id': 'dog', 'is_magic': false},
    {'id': 'cat', 'is_magic': false},
    {'id': 'unicorn', 'is_magic': true}
  ]
}
```

Could be loaded into the REPL with `animals` and `types` bound list of `struct` values.

The REPL could be started up with:

```
$ ./gradlew :cli:run -q --console=plain --args='-e config.sql'
```

(Note that shell expansions such as ~ do not work within the value of the `args` argument.)

Or if you have extracted one of the compressed archives:

```
$ ./bin/partiql -e config.sql
```

Expressions can then use the environment defined by `config.sql`:

```
PartiQL> SELECT name, type, is_magic FROM animals, types WHERE type = id
   |
==='
<<
  {
    'name': 'Kumo',
    'type': 'dog',
    'is_magic': false
  },
  {
    'name': 'Mochi',
    'type': 'dog',
    'is_magic': false
  },
  {
    'name': 'Lilikoi',
```



```
      'type': 'unicorn',
      'is_magic': true
    }
  >>
  ---
  OK!
```

To see the current REPL environment you can use `!global_env`, for example for the file above:

```
PartiQL> !global_env
   |
===' 
{
  'types': [
    {
      'id': 'dog',
      'is_magic': false
    },
    {
      'id': 'cat',
      'is_magic': false
    },
    {
      'id': 'unicorn',
      'is_magic': true
    }
  ],
  'animals': [
    {
      'name': 'Kumo',
      'type': 'dog'
    },
    {
      'name': 'Mochi',
      'type': 'dog'
    },
    {
      'name': 'Lilikoi',
      'type': 'unicorn'
    }
  ]
}
---
```



```
OK!
```

You can also add new values to the global environment or replace existing values using `!add_to_global_env`. The example below replaces the value bound to `types`

```
PartiQL> !add_to_global_env {'types': []}
   |
==='
{
  'types': []
}
---
OK!
PartiQL> !global_env
   |
==='
{
  'types': [],
  'animals': [
    {
      'name': 'Kumo',
      'type': 'dog'
    },
    {
      'name': 'Mochi',
      'type': 'dog'
    },
    {
      'name': 'Lilikoi',
      'type': 'unicorn'
    }
  ]
}
---
OK!
```

8 Working with Structure

Let's consider the following initial environment:

```
{
```



```
  'stores':[
    {
     'id': 5,
     'books': [
       {'title':'A', 'price': 5.0, 'categories':['sci-fi', 'action']},
       {'title':'B', 'price': 2.0, 'categories':['sci-fi', 'comedy']},
       {'title':'C', 'price': 7.0, 'categories':['action', 'suspense']},
       {'title':'D', 'price': 9.0, 'categories':['suspense']}
     ]
    },
    {
     'id': 6,
     'books': [
       {'title':'A', 'price': 5.0, 'categories':['sci-fi', 'action']},
       {'title':'E', 'price': 9.5, 'categories':['fantasy', 'comedy']},
       {'title':'F', 'price': 10.0, 'categories':['history']}
     ]
    }
  ]
}
```

Set the environment as below

```
PartiQL> !add_to_global_env { 'stores':[ { 'id': 5, 'books': [ {'title':'A', 'price': 5.0,
    'categories':['sci-fi', 'action']}, {'title':'B', 'price': 2.0, 'categories':['sci-fi',
     'comedy']}, {'title':'C', 'price': 7.0, 'categories':['action', 'suspense']}, {'title'
    :'D', 'price': 9.0, 'categories':['suspense']} ] }, { 'id': 6, 'books': [ {'title':'A',
     'price': 5.0, 'categories':['sci-fi', 'action']}, {'title':'E', 'price': 9.5, '
    categories':['fantasy', 'comedy']}, {'title':'F', 'price': 10.0, 'categories':['history
    ']} ] } ] }
```

If we wanted to find all books *as their own rows* with a price greater than 7 we can use paths on the `FROM` for this:

```
PartiQL> SELECT * FROM stores[*].books[*] AS b WHERE b.price > 7
   |
===' 
<<
  {
    'title': 'D',
    'price': 9.0,
    'categories': [
      'suspense'
```

```
      ]
    },
    {
      'title': 'E',
      'price': 9.5,
      'categories': [
        'fantasy',
        'comedy'
      ]
    },
    {
      'title': 'F',
      'price': 10.0,
      'categories': [
        'history'
      ]
    }
  >>
  ---
  OK!
```

If you wanted to also de-normalize the store ID and title into the above rows:

```
PartiQL> SELECT s.id AS store, b.title AS title FROM stores AS s, @s.books AS b WHERE b.
    price > 7
   |
===' 
<<
  {
    'store': 5,
    'title': 'D'
  },
  {
    'store': 6,
    'title': 'E'
  },
  {
    'store': 6,
    'title': 'F'
  }
>>
---
OK!
```

We can also use sub-queries with paths to predicate on sub-structure without changing the cardinality. So if we wanted to find all stores with books having prices greater than `9.5`

```
PartiQL> SELECT * FROM stores AS s
   | WHERE EXISTS(
   |    SELECT * FROM @s.books AS b WHERE b.price > 9.5
   | )
   |
==='
<<
  {
    'id': 6,
    'books': [
      {
        'title': 'A',
        'price': 5.0,
        'categories': [
          'sci-fi',
          'action'
        ]
      },
      {
        'title': 'E',
        'price': 9.5,
        'categories': [
          'fantasy',
          'comedy'
        ]
      },
      {
        'title': 'F',
        'price': 10.0,
        'categories': [
          'history'
        ]
      }
    ]
  }
>>
---
OK!
```


9 Reading/Writing Files

The REPL provides the read_file function to stream data from a file. The files needs to be placed in the folder cli. For example:

Create a file called data.ion in the cli folder with the following contents

```
{ 'city': 'Seattle', 'state': 'WA' }
{ 'city': 'Bellevue', 'state': 'WA' }
{ 'city': 'Honolulu', 'state': 'HI' }
{ 'city': 'Rochester', 'state': 'NY' }
```

Select the cities that are in HI and NY states

```
PartiQL> SELECT city FROM read_file('data.ion') AS c, `["HI", "NY"]` AS s WHERE c.state = s
   |
==='
<<
  {
    'city': 'Honolulu'
  },
  {
    'city': 'Rochester'
  }
>>
------
OK!
```

The REPL also has the capability to write files with the write_file function:

```
PartiQL> write_file('out.ion', SELECT * FROM _)
   |
==='
true
------
OK!
```

A file called out.ion will be created in the cli directory with the following contents

```
{
  city:Honolulu
}
{
  city:Rochester
```



```
}
```

Functions and expressions can be used in the *global configuration* as well. Consider the following `config.ion`:

```
{
  'data': read_file('data.ion')
}
```

The `data` variable will now be bound to file containing Ion:

```
PartiQL> SELECT * FROM data
   |
==='
<<
    {
      'city: ;Seattle;,
      'state: 'WA;
    },
    {
      'city: 'Bellevue',
      'state: 'WA'
    },
    {
      'city: 'Honolulu',
      'state: 'HI'
    },
    {
      'city: 'Rochester',
      'state: 'NY'
    }
>>
------
OK!
```

10 TSV/CSV Data

The `read_file` function supports an optional struct argument to add additional parsing options. Parsing delimited files can be specified with the `type` field with a string `tsv` or `csv` to parse tab or comma separated values respectively.

Create a file called `simple.csv` in the `cli` directory with the following contents


```
title,category,price
harry potter,book,7.99
dot,electronics,49.99
echo,electronics,99.99
```

```
PartiQL> read_file('simple.csv', {'type':'csv'})
   |
==='
<<
    {
      _0:'title',
      _1:'category',
      _2:'price'
    },
    {
      _0:'harry potter',
      _1:'book',
      _2:'7.99'
    },
    {
      _0:'dot',
      _1:'electronics',
      _2:'49.99'
    },
    {
      _0:'echo',
      _1:'electronics',
      _2:'99.99'
    }
>>
----
OK!
```

The options `struct` can also define if the first row for delimited data should be the column names with the `header` field.

```
PartiQL> read_file('simple.csv', {'type': 'csv', 'header': true})
   |
==='
<<
    {
      'title': 'harry potter',
```

```
        'category': 'book',
        'price': '7.99'
      },
      {
        'title': 'dot',
        'category': 'electronics',
        'price': '49.99'
      },
      {
        'title': 'echo',
        'category': 'electronics',
        'price': '99.99'
      }
    >>
    ----
    OK!
```

Auto conversion can also be specified numeric and timestamps in delimited data.

```
    PartiQL> read_file('simple.csv', {'type':'csv', 'header':true, 'conversion':'auto'})
       |
    ==='
    <<
      {
        'title':' harry potter',
        'category': 'book',
        'price': 7.99
      },
      {
        'title: 'dot',
        'category': 'electronics',
        'price': 49.99
      },
      {
        'title: 'echo',
        'category': 'electronics',
        'price': 99.99
      }
    >>
    ----
    OK!
```

Writing TSV/CSV data can be done by specifying the optional `struct` argument to specify output format to the `write_file` function. Similar to the `read_file` function, the `type` field can be used to specify `tsv`, `csv`, or `ion` output.

```
PartiQL> write_file('out.tsv', {'type':'tsv'}, SELECT name, type FROM animals)
   |
==='
true
----
OK!
```

This would produce the following file:

```
$ cat out.tsv
Kumo    dog
Mochi   dog
Lilikoi unicorn
```

The options `struct` can also specify a `header` Boolean field to indicate whether the output TSV/CSV should have a header row.

```
PartiQL> write_file('out.tsv', {'type':'tsv', 'header':true}, SELECT name, type FROM
    animals)
   |
==='
true
----
OK!
```

Which would produce the following file:

```
$ cat out.tsv
name    type
Kumo    dog
Mochi   dog
Lilikoi unicorn
```

11 PartiQL Tutorial

TBD

12 PartiQL User Guide

12.1 Introduction

This is the PartiQL implementation's user guide. The goal of this document is to provide to users of PartiQL information on the features implemented and any deviation from the PartiQL specification.

12.2 Data Types

12.2.1 Decimal

PartiQL decimals are based on Ion decimals from the Ion Specification[1] but with a maximum precision of 38 digits, numbers outside this precision range will be rounded using a round half even strategy. Examples:

```
1.00000000000000000000000000000000000000000001 -> 1.0000000000000000000000000000000000000
1.99999999999999999999999999999999999999999999 -> 2.0000000000000000000000000000000000000
```

12.3 Built-in Functions

This section provides documentation for all built-in functions available with the reference implementation. For each function the documentation provides

1. A one sentence explanation of the functions intent.
2. The function's **signature** that specifies data types and names for each input argument, and, the expected data type for the function's return value. A function signature consists of the function's name followed by a colon : then a space separated list of data types–one for each formal argument of the function–followed by an arrow -> followed by the function's return type, e.g., `add: Integer Integer -> Integer` is the signature for `add` which accepts 2 inputs both `Integer`'s and returns one value of type `Integer`.
3. The function's **header** that specifies names for each of the function's formal arguments. Any documentation following the header can refer to the function's formal arguments by name.
4. The function's **purpose statement** that further expands on how the function behaves and specifies any pre- and/or post-conditions.
5. A list of examples calling the function and their expected results.

12.3.1 Unknown (`null` and `missing`) propagation

Unless otherwise stated all functions listed below propagate `null` and `missing` argument values. Propagating `null` and `missing` values is defined as: if any function argument is either `null` or `missing` the function will return `null`, e.g.,

```
CHAR_LENGTH(null)    -- `null`
CHAR_LENGTH(missing) -- `null` (also returns `null`)
```

12.3.2 CAST

Given a value and a target data type, attempt to coerce the value to the target data type.

Signature `CAST: Any DataType -> DataType`

Where `DataType` is one of

- `missing`
- `null`
- `integer`
- `boolean`
- `float`
- `decimal`
- `timestamp`
- `symbol`
- `string`
- `list`
- `struct`
- `bag`

Header `CAST(exp AS dt)`

Purpose Given an expression, `exp` and the data type name, `dt`, evaluate `expr` to a value, `v` and alter the data type of `v` to `DT(dt)`. If the conversion cannot be made the implementation signals an error.

The runtime support for casts is

- Casting to `null` from
  - `null` is a no-op
  - `missing` returns `null`
  - else error
- Casting to `missing` from
  - `missing` is a no-op
  - `null` returns `missing`
  - else error
- Casting to `integer` from
  - Integer: is a no-op

  - Boolean: `true` returns 1, `false` returns 0
  - String or Symbol: attempt to parse the content as an Integer and return the Integer, else error.
  - Float or Decimal: gets narrowed to Integer
  - else error
- Casting to `boolean` from
  - Boolean is a no-op
  - Integer or Decimal or Float: if `v` is a representation of the number `0` (e.g., `0` or `-0` or `0e0` or `0d0` et.) then `false` else `true`
  - String or Symbol: `true` unless `v` matches–ignoring character case–the Ion string `"false"` or matches–ignoring character case– the Ion symbol `'false'` then return `false`
  - else error
- Casting to `float` from
  - Float is a no-op
  - Boolean: `false` return `0.0`, `true` returns `1.0`
  - Integer or Decimal: convert to Float and return
  - String or Symbol: attempt to parse as Float and return the Float value, else error
  - else error
- Casting to `decimal` from
  - Decimal is a no-op
  - Boolean: return `1d0` if `true`, `0d0` if `false`
  - String or Symbol: attemp to parse as Decimal and return Decimal value, else error
  - else error
- Casting to `timestamp` from
  - Timestamp is a no-op
  - String or Symbol: attemp to parse as Timestamp and return the Timestamp value, else error
  - else error
- Casting to `symbol` from
  - Symbol is a no-op
  - Integer or Float or Decimal: narrow to Integer and return the value as a Symbol, i.e, a Symbol with the same sequence of digits as characters
  - String: return the String as a Symbol, i.e., represent the same sequence of characters as a Symbol
  - Boolean: return `'true'` for `true` and `'false'` for `false`
  - Timestamp: return the Symbol representation of the Timestamp, i.e., represent the same sequence of digits and characters as a Symbol
  - else error
- Casting to `string` from
  - String is a no-op

  - Integer or Float or Decimal: narrow to Integer and return the value as a String, i.e, a String with the same sequence of digits as characters
  - Symbol: return the String as a Symbol, i.e., represent the same sequence of characters as a String
  - Boolean: return "`true`" for `true` and "`false`" for `false`
  - Timestamp: return the String representation of the Timestamp, i.e., represent the same sequence of digits and characters as a String
  - else error
- Casting to `list` from
  - List is a no-op
  - Bag: return a list with the same elements. The order of the elements in the resulting list is unspecified.
  - else error
- Casting to `struct` from
  - Struct is a no-op
  - else error
- Casting to `bag` from
  - Bag is a no-op
  - List: return a list with the same elements. The order of the elements in the resulting bag is unspecified.

Examples :

```
-- Unknowns propagation
CAST(null    AS null)    -- null
CAST(missing AS null)    -- null
CAST(missing AS missing) -- null
CAST(null    AS missing) -- null
CAST(null    AS boolean) -- null (null AS any data type name result to null)
CAST(missing AS boolean) -- null (missing AS any data type name result to null)

-- any value that is not an unknown cannot be cast to `null` or `missing`
CAST(true AS null)    -- error
CAST(true AS missing) -- error
CAST(1    AS null)    -- error
CAST(1    AS missing) -- error

-- AS boolean
CAST(true      AS boolean) -- true no-op
CAST(0         AS boolean) -- false
CAST(1         AS boolean) -- true
```



```
CAST(`1e0`     AS boolean) -- true (float)
CAST(`1d0`     AS boolean) -- true (decimal)
CAST('a'       AS boolean) -- false
CAST('true'    AS boolean) -- true (PartiQL string 'true')
CAST(`'true'`  AS boolean) -- true (Ion symbol `'true'`)
CAST(`'false'` AS boolean) -- false (Ion symbol `'false'`)

-- AS integer
CAST(true   AS integer) -- 1
CAST(false  AS integer) -- 0
CAST(1      AS integer) -- 1
CAST(`1d0`  AS integer) -- 1
CAST(`1d3`  AS integer) -- 1000
CAST(1.00   AS integer) -- 1
CAST('12'   AS integer) -- 12
CAST('aa'   AS integer) -- error
CAST(`'22'` AS integer) -- 22
CAST(`'x'`  AS integer) -- error

-- AS flaot
CAST(true   AS float) -- 1e0
CAST(false  AS float) -- 0e0
CAST(1      AS float) -- 1e0
CAST(`1d0`  AS float) -- 1e0
CAST(`1d3`  AS float) -- 1000e0
CAST(1.00   AS float) -- 1e0
CAST('12'   AS float) -- 12e0
CAST('aa'   AS float) -- error
CAST(`'22'` AS float) -- 22e0
CAST(`'x'`  AS float) -- error

-- AS decimal
CAST(true   AS decimal) -- 1.
CAST(false  AS decimal) -- 0.
CAST(1      AS decimal) -- 1.
CAST(`1d0`  AS decimal) -- 1. (REPL printer serialized to 1.)
CAST(`1d3`  AS decimal) -- 1d3
CAST(1.00   AS decimal) -- 1.00
CAST('12'   AS decimal) -- 12.
CAST('aa'   AS decimal) -- error
CAST(`'22'` AS decimal) -- 22.
CAST(`'x'`  AS decimal) -- error
```



```
-- AS timestamp
CAST(`2001T`                       AS timestamp) -- 2001T
CAST('2001-01-01T'                 AS timestamp) -- 2001-01-01
CAST(`'2010-01-01T00:00:00.000Z'` AS timestamp) -- 2010-01-01T00:00:00.000Z
CAST(true                          AS timestamp) -- error
CAST(2001                          AS timestamp) -- error

-- AS symbol
CAST(`'xx'`                      AS symbol) -- xx (`'xx'` is an Ion symbol)
CAST('xx'                        AS symbol) -- xx ('xx' is a string)
CAST(42                          AS symbol) -- '42'
CAST(`1e0`                       AS symbol) -- '1'
CAST(`1d0`                       AS symbol) -- '1'
CAST(true                        AS symbol) -- 'true'
CAST(false                       AS symbol) -- 'false'
CAST(`2001T`                     AS symbol) -- '2001T'
CAST(`2001-01-01T00:00:00.000Z` AS symbol) -- '2001-01-01T00:00:00.000Z`

-- AS string
CAST(`'xx'`                      AS string) -- "xx" (`'xx'` is an Ion symbol)
CAST('xx'                        AS string) -- "xx" ('xx' is a string)
CAST(42                          AS string) -- "42"
CAST(`1e0`                       AS string) -- "1.0"
CAST(`1d0`                       AS string) -- "1"
CAST(true                        AS string) -- "true"
CAST(false                       AS string) -- "false"
CAST(`2001T`                     AS string) -- "2001T"
CAST(`2001-01-01T00:00:00.000Z` AS string) -- "2001-01-01T00:00:00.000Z"

-- AS struct
CAST(`{ a: 1 }` AS struct) -- { a:1 }
CAST(true       AS struct) -- err

-- AS list
CAST(`[1, 2, 3]`        AS list) -- [ 1, 2, 3 ] (REPL does not diplay the parens and commas
    )
CAST(<<'a', { 'b':2 }>> AS list) -- [ a, { 'b':2 } ] (REPL does not diplay the parens and
    commas)
CAST({ 'b':2 }          AS list) -- error

-- AS bag
CAST([1,2,3]      AS bag) -- <<1,2,3>> (REPL does not display << >> and commas)
CAST([1,[2],3]    AS bag) -- <<1,[2],3>> (REPL does not display << >> and commas)
```

```
CAST(<<'a', 'b'>> AS bag) -- <<'a', 'b'>> (REPL does not display << >> and commas)
```

12.3.3 CHAR_LENGTH, CHARACTER_LENGTH

Counts the number of characters in the specified string, where ‘character’ is defined as a single unicode code point.

Note: `CHAR_LENGTH` and `CHARACTER_LENGTH` are synonyms.

Signature `CHAR_LENGTH: String -> Integer`

`CHARACTER_LENGTH: String -> Integer`

Header `CHAR_LENGTH(str)`

`CHARACTER_LENGTH(str)`

Purpose Given a `String` value `str` return the number of characters (code points) in `str`.

Examples

```
CHAR_LENGTH('')          -- 0
CHAR_LENGTH('abcdefg')   -- 7
CHAR_LENGTH('😁😞😸😸') -- 4 (non-BMP unicode characters)
CHAR_LENGTH('e̫')         -- 2 (because 'e̫' is two codepoints: the letter 'e' and
    combining character U+032B)
```

12.3.4 COALESCE

Evaluates the arguments in order and returns the first non unknown, i.e. first non-`null` or non-`missing`. This function does **not** propagate `null` and `missing`.

Signature `COALESCE: Any Any ... -> Any`

Header `COALESCE(exp, [exp ...])`

Purpose Given a list of 1 or more arguments, evaluates the arguments left-to-right and returns the first value that is **not** an unknown (`missing` or `null`).

Examples

```
COALESCE(1)                -- 1
COALESCE(null)             -- null
COALESCE(null, null)       -- null
COALESCE(missing)          -- null
COALESCE(missing, missing) -- null
COALESCE(1, null)          -- 1
```

```
COALESCE(null, null, 1)     -- 1
COALESCE(null, 'string')    -- 'string'
COALESCE(missing, 1)        -- 1
```

12.3.5 DATE_ADD

Given a data part, a quantity and a timestamp, returns an updated timestamp by altering date part by quantity

Signature `DATE_ADD: DatePart Integer Timestamp -> Timestamp`

Where `DatePart` is one of

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`

Header `DATE_ADD(dp, q, timestamp)`

Purpose Given a data part `dp`, a quantity `q`, and, an Ion timestamp `timestamp` returns an updated timestamp by applying the value for `q` to the `dp` component of `timestamp`. Positive values for `q` add to the `timestamp`'s `dp`, negative values subtract.

The value for `timestamp` as well as the return value from `DATE_ADD` must be a valid Ion Timestamp

Examples

```
DATE_ADD(year, 5, `2010-01-01T`)                    -- 2015-01-01 (equivalent to 2015-01-01T)
DATE_ADD(month, 1, `2010T`)                         -- 2010-02T (result will add precision as
    necessary)
DATE_ADD(month, 13, `2010T`)                        -- 2011-02T
DATE_ADD(day, -1, `2017-01-10T`)                    -- 2017-01-09 (equivalent to 2017-01-09T)
DATE_ADD(hour, 1, `2017T`)                          -- 2017-01-01T01:00-00:00
DATE_ADD(hour, 1, `2017-01-02T03:04Z`)              -- 2017-01-02T04:04Z
DATE_ADD(minute, 1, `2017-01-02T03:04:05.006Z`) -- 2017-01-02T03:05:05.006Z
DATE_ADD(second, 1, `2017-01-02T03:04:05.006Z`) -- 2017-01-02T03:04:06.006Z
```

12.3.6 DATE_DIFF

Given a date part and two valid timestamps returns the difference in date parts.

Signature `DATE_DIFF: DatePart Timestamp Timestamp -> Integer`

See DATE_ADD for the definition of `DatePart`

Header `DATE_DIFF(dp, t1, t2)`

Purpose Given a date part `dp` and two timestamps `t1` and `t2` returns the difference in value for `dp` part of `t1` with `t2`. The return value is a negative integer when the `dp` value of `t1` is greater than the `dp` value of `t2`, and, a positive integer when the `dp` value of `t1` is less than the `dp` value of `t2`.

Examples

```
DATE_DIFF(year, `2010-01-01T`, `2011-01-01T`)            -- 1
DATE_DIFF(year, `2010T`, `2010-05T`)                     -- 4 (2010T is equivalent to
    2010-01-01T00:00:00.000Z)
DATE_DIFF(month, `2010T`, `2011T`)                       -- 12
DATE_DIFF(month, `2011T`, `2010T`)                       -- -12
DATE_DIFF(day, `2010-01-01T23:00T`, `2010-01-02T01:00T`) -- 0 (need to be at least 24h
    apart to be 1 day apart)
```

12.3.7 EXISTS

Given an PartiQL value returns **`true`** if and only if the value is a non-empty sequence, returns **`false`** otherwise.

Signature `EXISTS: Any -> Boolean`

Header `EXISTS(val)`

Purpose Given an PartiQL value, `val`, returns **`true`** if and only if `val` is a non-empty sequence, returns **`false`** otherwise. This function does **not** propagate **`null`** and `missing`.

Examples

```
EXISTS(`[]`)        -- false (empty list)
EXISTS(`[1, 2, 3]`) -- true (non-empty list)
EXISTS(`[missing]`) -- true (non-empty list)
EXISTS(`{}`)        -- false (empty struct)
EXISTS(`{ a: 1 }`)  -- true (non-empty struct)
EXISTS(`()`)        -- false (empty s-expression)
EXISTS(`(+ 1 2)`)   -- true (non-empty s-expression)
EXISTS(`<<>>`)      -- false (empty bag)
EXISTS(`<<null>>`)  -- true (non-empty bag)
EXISTS(1)           -- false
EXISTS(`2017T`)     -- false
EXISTS(null)        -- false
EXISTS(missing)     -- false
```


12.3.8 EXTRACT

Given a date part and a timestamp returns then timestamp's date part value.

Signature `EXTRACT: ExtractDatePart Timestamp -> Integer`

where `ExtractDatePart` is one of

- `year`
- `month`
- `day`
- `hour`
- `minute`
- `second`
- `timezone_hour`
- `timezone_minute`

Note that `ExtractDatePart` **differs** from `DatePart` in DATE_ADD.

Header `EXTRACT(edp FROM t)`

Purpose Given a date part, `edp`, and a timestamp `t` return `t`'s value for `edp`. This function allows for `t` to be unknown (`null` or `missing`) but **not** `edp`. If `t` is unknown the function returns `null`.

Examples

```
EXTRACT(YEAR FROM `2010-01-01T`)                           -- 2010
EXTRACT(MONTH FROM `2010T`)                                -- 1 (equivalent to 2010-01-01
    T00:00:00.000Z)
EXTRACT(MONTH FROM `2010-10T`)                             -- 10
EXTRACT(HOUR FROM `2017-01-02T03:04:05+07:08`)             -- 3
EXTRACT(MINUTE FROM `2017-01-02T03:04:05+07:08`)           -- 4
EXTRACT(TIMEZONE_HOUR FROM `2017-01-02T03:04:05+07:08`)    -- 7
EXTRACT(TIMEZONE_MINUTE FROM `2017-01-02T03:04:05+07:08`)  -- 8
```

12.3.9 LOWER

Given a string convert all upper case characters to lower case characters.

Signature `LOWER: String -> String`

Header `LOWER(s)`

Purpose Given a string, `s`, alter every upper case character in `s` to lower case. Any non-upper cased characters remain unchanged. This operation does rely on the locale specified by the runtime configuration. The implementation, currently, relies on Java's String.toLowerCase() documentation.

Examples

```
LOWER('AbCdEfG!@#$') -- 'abcdefg!@#$'
```

12.3.10 SIZE

Given any container data type (i.e., list, structure or bag) return the number of elements in the container.

Signature `SIZE: Container -> Integer`

Header `SIZE(c)`

Purpose Given a container, `c`, return the number of elements in the container. If the input to `SIZE` is not a container the implementation throws an error.

Examples

```
SIZE(`[]`)                   -- 0
SIZE(`[null]`)               -- 1
SIZE(`[1,2,3]`)              -- 3
SIZE(<<'foo', 'bar'>>)       -- 2
SIZE(`{foo: bar}`)           -- 1 (number of key-value pairs)
SIZE(`[{foo: 1}, {foo: 2}]`) -- 2
SIZE(12)                     -- error
```

12.3.11 NULLIF

Given two expressions return **`null`** if the two expressions evaluate to the same value, else, returns the result of evaluating the first expression

Signature `NULLIF: Any Any -> Any`

Header `NULLIF(e1, e2)`

Purpose Given two expression, `e1` and `e2`, evaluate both expression to get `v1` and `v2` respectively, and return **`null`** if and only if `v1` equals `v2`, else, return `v1`. The implementation of `NULLIF` uses `=` for equality, i.e., `v1` and `v2` are considered equal by `NULLIF` if and only if `v1 = v2` is true.

Note, `NULLIF` does **not** propagate unknowns (**`null`** and `missing`).

Examples

```
NULLIF(1, 1)              -- null
NULLIF(1, 2)              -- 1
NULLIF(1.0, 1)            -- null
NULLIF(1, '1')            -- 1
NULLIF([1], [1])          -- null
NULLIF(1, NULL)           -- 1
```

```
NULLIF(NULL, 1)          -- null
NULLIF(null, null)       -- null
NULLIF(missing, null)    -- null
NULLIF(missing, missing) -- null
```

12.3.12 SUBSTRING

Given a string, a start index and optionally a length, returns the substring from the start index up to the end of the string, or, up to the length provided.

Signature `SUBSTRING: String Integer [ NNegInteger ] -> String`

Where `NNegInteger` is a non-negative integer, i.e., 0 or greater.

Header `SUBSTRING(str, start [ , length ])`

`SUBSTRING(str FROM start [ FOR length ])`

Purpose Given a string, `str`, a start position, `start` and optionally a length, `length`, extract the characters (code points) starting at index `start` and ending at (`start` + `length`) - 1. If `length` is omitted, then proceed till the end of `str`.

The first character of `str` has index 1.

Examples

```
SUBSTRING("123456789", 0)      -- "123456789"
SUBSTRING("123456789", 1)      -- "123456789"
SUBSTRING("123456789", 2)      -- "23456789"
SUBSTRING("123456789", -4)     -- "123456789"
SUBSTRING("123456789", 0, 999) -- "123456789"
SUBSTRING("123456789", 0, 2)   -- "1"
SUBSTRING("123456789", 1, 999) -- "123456789"
SUBSTRING("123456789", 1, 2)   -- "12"
SUBSTRING("1", 1, 0)           -- ""
SUBSTRING("1", 1, 0)           -- ""
SUBSTRING("1", -4, 0)          -- ""
SUBSTRING("1234", 10, 10)      -- ""
```

12.3.13 TO_STRING

Given a timestamp and a format pattern return a string representation of the timestamp in the given format.

Signature `TO_STRING: Timestamp TimeFormatPattern -> String`

Where `TimeFormatPattern` is a String with the following special character interpretations

Format	Example	Description
`yy`	69	2-digit year
`y`	1969	4-digit year
`yyyy`	1969	Zero padded 4-digit year
`M`	1	Month of year
`MM`	01	Zero padded month of year
`MMM`	Jan	Abbreviated month year name
`MMMM`	January	Full month of year name
`MMMMM`	J	Month of year first letter (NOTE: not valid for use with to_timestamp function)
`d`	2	Day of month (1-31)
`dd`	02	Zero padded day of month (01-31)
`a`	AM	AM or PM of day
`h`	3	Hour of day (1-12)
`hh`	03	Zero padded hour of day (01-12)
`H`	3	Hour of day (0-23)
`HH`	03	Zero padded hour of day (00-23)
`m`	4	Minute of hour (0-59)
`mm`	04	Zero padded minute of hour (00-59)
`s`	5	Second of minute (0-59)
`ss`	05	Zero padded second of minute (00-59)
`S`	0	Fraction of second (precision: 0.1, range: 0.0-0.9)
`SS`	06	Fraction of second (precision: 0.01, range: 0.0-0.99)
`SSS`	060	Fraction of second (precision: 0.001, range: 0.0-0.999)
…	…	…
`SSSSSSSSS`	060000000	Fraction of second (maximum precision: 1 nanosecond, range: 0.0-0.999999999)

Format	Example	Description
`n`	60000000	Nano of second
`X`	+07 or Z	Offset in hours or “Z” if the offset is 0
`XX` or `XXXX`	+0700 or Z	Offset in hours and minutes or “Z” if the offset is 0
`XXX` or `XXXXX`	+07:00 or Z	Offset in hours and minutes or “Z” if the offset is 0
`x`	+07	Offset in hours
`xx` or `xxxx`	+0700	Offset in hours and minutes
`xxx` or `xxxxx`	+07:00	Offset in hours and minutes

Header `TO_STRING(t,f)`

Purpose Given a timestamp, `t`, and a format pattern, `f`, as a String, return a string representation of `t` in format `f`.

Examples

```
TO_STRING(`1969-07-20T20:18Z`,  'MMMM d, y')                  -- "July 20, 1969"
TO_STRING(`1969-07-20T20:18Z`, 'MMM d, yyyy')                 -- "Jul 20, 1969"
TO_STRING(`1969-07-20T20:18Z`, 'M-d-yy')                      -- "7-20-69"
TO_STRING(`1969-07-20T20:18Z`, 'MM-d-y')                      -- "07-20-1969"
TO_STRING(`1969-07-20T20:18Z`, 'MMMM d, y h:m a')             -- "July 20, 1969 8:18 PM"
TO_STRING(`1969-07-20T20:18Z`, 'y-MM-dd''T''H:m:ssX')         -- "1969-07-20T20:18:00Z"
TO_STRING(`1969-07-20T20:18+08:00Z`, 'y-MM-dd''T''H:m:ssX')   -- "1969-07-20T20:18:00Z"
TO_STRING(`1969-07-20T20:18+08:00`, 'y-MM-dd''T''H:m:ssXXXX')   -- "1969-07-20T20
    :18:00+0800"
TO_STRING(`1969-07-20T20:18+08:00`, 'y-MM-dd''T''H:m:ssXXXXX')  -- "1969-07-20T20
    :18:00+08:00"
```

12.3.14 TO_TIMESTAMP

Given a string convert it to a timestamp. This is the inverse operation of `TO_STRING`

Signature `TO_TIMESTAMP: String [ TimeFormatPattern ] -> Timestamp`

See definition of `TimeFormatPattern` in TO_STRING.

Header `TO_TIMESTAMP(str[ , f ])`

Purpose Given a string, `str`, and an optional format pattern, `f`, as a String return a timestamp whose values are extracted from `str` using `f`.

If the <`format pattern`> argument is omitted, <`string`> is assumed to be in the format of a standard Ion timestamp. This is the only recommended way to parse an Ion timestamp using this function.

Zero padding is optional when using a single format symbol (e.g. `y`, `M`, `d`, `H`, `h`, `m`, `s`) but required for their zero padded variants (e.g. `yyyy`, `MM`, `dd`, `HH`, `hh`, `mm`, `ss`).

Special treatment is given to 2-digit years (format symbol `yy`). 1900 is added to values greater than or equal to 70 and 2000 is added to values less than 70.

Month names and AM/PM specifiers are case-insensitive.

Examples

Single argument parsing an Ion timestamp:

```
TO_TIMESTAMP('2007T')                         -- `2007T`
TO_TIMESTAMP('2007-02-23T12:14:33.079-08:00') -- `2007-02-23T12:14:33.079-08:00`
TO_TIMESTAMP('2016', 'y')                     -- `2016T`
TO_TIMESTAMP('2016', 'yyyy')                  -- `2016T`
TO_TIMESTAMP('02-2016', 'MM-yyyy')            -- `2016-02T`
TO_TIMESTAMP('Feb 2016', 'MMM yyyy')          -- `2016-02T`
TO_TIMESTAMP('Febrary 2016', 'MMMM yyyy')     -- `2016-02T`
```

Notes:

All issues for PartiQL's `TO_TIMESTAMP` function.

Internally, this is implemented with Java 8's `java.time` package. There are a few differences between Ion's timestamp and the `java.time` package that create a few hypothetically infrequently encountered caveats that do not really have good workarounds at this time.

- The Ion specification allows for explicitly signifying an unknown timestamp with a negative zero offset (i.e. the `-00:00` at the end of `2007-02-23T20:14:33.079-00:00`) but Java 8's `DateTimeFormatter` doesn't recognize this. **Hence, unknown offsets specified in this manner will be parsed as if they had an offset of `+00:00`, i.e. UTC.** To avoid this issue when parsing Ion formatted timestamps, use the single argument variant of `TO_TIMESTAMP`. There is no workaround for custom format patterns at this time.
- `DateTimeFormatter` is capable of parsing UTC offsets to the precision of seconds, but Ion Timestamp's precision for offsets is minutes. TimestampParser currently handles this by throwing an exception when an attempt is made to parse a timestamp with an offset that does does not land on a minute boundary. For example, parsing this timestamp would throw an exception: `May 5, 2017 8:52pm +08:00:01` while `May 5, 2017 8:52pm +08:00:00` would not.
- Ion Java's Timestamp allows specification of offsets up to +/- 23:59, while an exception is thrown by `DateTimeFormatter` for any attempt to parse an offset greater than +/- 18:00. For example, attempting to parse: `May 5, 2017 8:52pm +18:01` would cause and exception to be thrown. (Note: the Ion

specification does indicate minimum and maximum allowable values for offsets.) In practice this will not be an issue for systems that do not abuse the offset portion of Timestamp because real-life offsets do not exceed +/- 12h.

12.3.15 TRIM

Trims leading and/or trailing characters from a String.

Signature `TRIM: [ String ] String -> String`

Header `TRIM([[LEADING|TRAILING|BOTH r] FROM] str)`

Purpose Given a string, `str`, and an optional *set* of characters to remove, `r`, specified as a string, return the string with any character from `r` found at the beginning or end of `str` removed.

If `r` is not provided it defaults to `' '`.

Examples

```
TRIM('       foobar         ')               -- 'foobar'
TRIM('      \tfoobar\t         ')            -- '\tfoobar\t'
TRIM(LEADING FROM '       foobar         ')  -- 'foobar         '
TRIM(TRAILING FROM '       foobar         ') -- '       foobar'
TRIM(BOTH FROM '       foobar         ')     -- 'foobar'
TRIM(BOTH '😁' FROM '😁😁😁😁foobar')          -- 'foobar'
TRIM(BOTH '12' FROM '1112211foobar22211122') -- 'foobar'
```

12.3.16 UPPER

Given a string convert all lower case characters to upper case characters.

Signature `UPPER: String -> String`

Header `UPPER(str)`

Purpose Given a string, `str`, alter every upper case character is `str` to lower case. Any non-lower cases characters remain unchanged. This operation does rely on the locale specified by the runtime configuration. The implementation, currently, relies on Java's String.toLowerCase() documentation.

Examples

```
UPPER('AbCdEfG!@#$') -- 'ABCDEFG!@#$'
```

12.3.17 UTCNOW

Returns the current time in UTC as a timestamp.

Signature `UTCNOW: -> Timestamp`

Header `UTCNOW()`

Purpose Return the current time in UTC as a timestamp

Examples

```
UTCNOW() -- 2017-10-13T16:02:11.123Z
```

References

[1] Ion Committee, "Ion Specification 1.0," 2009. [Online]. Available: https://amzn.github.io/ion-docs/spec.html.

