Redshift User Defined Functions in Python

Periscope Data

April 1, 2015

https://www.periscopedata.com/blog/redshift-user-defined-functions-python

Today the Redshift team announced support for User Defined Functions! UDFs allow you to create your own SQL functions, backed by the power and flexibility of Python. Now you can add your favorite functions from other databases, or invent new ones, to make your data analysis much easier.

Based on what we've seen customers doing in Redshift, we put together a few dozen helpful UDFs for everyone to use. They are hosted on GitHub. Here's what's available so far:

New Redshift Functions from Periscope

JSON Array Support

- `json_array_first` - get the first element of the array
- `json_array_last` - get the last element of the array
- `json_array_nth` - gets the Nth element in the array

- `json_array_sort` - sorts the elements in the array
- `json_array_reverse` - reverses the order of elements in the array
- `json_array_pop` - returns the array without the last element
- `json_array_push` - returns the array with the new element at the end
- `json_array_concat` - combines two arrays into one

MySQL Date Compatibility Helpers

- `mysql_year` - extracts the year from the timestamp
- `mysql_month` - extracts the month (1-12) from the timestamp
- `mysql_day` - extracts the date (1-31) from the timestamp
- `mysql_hour` - extracts the hour (0-23) from the timestamp
- `mysql_minute` - extracts the minute from the timestamp
- `mysql_second` - extracts the second from the timestamp
- `mysql_yearweek` - formats a timestamp as a year-week (YYYYWW)

Number Utilities

- `format_num` - formats a number with commas, percents, padding, etc.
- `second_max` - gets the second-highest value from a numeric column

Varchar Utilities

- `email_name` - extracts the mailbox name from the email address
- `email_domain` - extracts the domain from the email address
- `url_protocol` - extracts the protocol from the URL

- `url_domain` - extracts the domain from the URL
- `url_path` - extract the path from the URL
- `url_param` - extract a parameter's value from the URL
- `split_count` - split a varchar into parts and get the count of those parts
- `titlecase` - title-case the varchar value
- `str_multiply` - repeats the varchar several times
- `str_index` - find the leftmost index of a substring in the varchar
- `str_rindex` - find the rightmost index of a substring in the varchar
- `str_count` - counts the number of occurrences of a substring within the varchar

Time Helpers

- `now` - a common alias for `getdate`
- `posix_timestamp` - get the number of seconds since the epoch for a timestamp

All of these UDFs were built using a framework we created to make managing and testing UDFs easy.

Installing, Developing and Contributing UDFs

To make development easier, Periscope built a framework for developing UDFs. The framework is available in our GitHub repo. Creating a new UDF is as simple as adding a few lines to a config file.

Here's the config to make a UDF called `email_domain`, it extracts the domain from an email address. The config defines the name, input and output types, the Python function body, and a several unit tests:

```
{
  type:        :function,
  name:        :email_domain,
  description: "Gets the domain from the email address",
  params:      "email varchar(max)",
  return_type: "varchar(max)",
  body:        %~
    if not email:
      return None
    return email.split('@')[-1]
  ~,
  tests:  [
    { query: "select ?('sam@company.com')",
      expect: 'company.com',
      example: true
    }, {
      query: "select ?('alex@othercompany.com')",
      expect: 'othercompany.com',
      example: true
    },
  ]
```

```
}
```

And the `udf.rb` runner takes care of creating, dropping, testing, and pretty-printing the UDF's SQL:

```
Usage:
  ruby udf.rb <action> [udf_name]
Actions:
  load    Loads UDFs into your database
  drop    Removes UDFs into your database
  test    Runs UDF unit tests on your database
  print   Pretty-print SQL for making the UDFs
Examples:
  ruby udf.rb load
  ruby udf.rb drop url_domain
  ruby udf.rb test json_array_first
  ruby udf.rb print
```

For example, to load this UDF, run:

```
ruby udf.rb load email_domain
```

To test this UDF, run:

```
ruby udf.rb test email_domain
```

And you'll see output like this:

```
Making function email_domain

Testing function email_domain....
```

With this framework it only takes a couple minutes to make a UDF, the unit tests prevent the UDFs from breaking, and the automation makes adding UDFs to new clusters a breeze.

We hope you'll find this framework useful when creating and maintaining UDFs, and look forward to feature requests, improvements, and many more UDFs!

To better understand how UDFs work, let's dive in to the syntax for creating them.

How To Write Scalar UDFs

The first kind of UDF supported by Redshift are Scalar UDFs. They apply to individual values. They work like a lot of the functions you're used to: `date_trunc`, `json_extract_path_text`, `getdate`, `round`, etc. Here's a UDF to get the domain from an email address:

```
/*

EMAIL_DOMAIN

Gets the domain from the email address

Examples:

  select email_domain('sam@company.com')

    --> 'company.com'

  select email_domain('alex@othercompany.com')
```

```
    --> 'othercompany.com'
*/
create or replace function email_domain (email varchar(max))
  returns varchar(max)
  stable as $$
    if not email:
      return None
    return email.split('@')[-1]
  $$ language plpythonu
```

The function is defined by `create or replace email_domain`, the first and only paramater is a `varchar` called `email`, and the return type is also a `varchar`.

There are three kinds of function types and they are used by the query planner to know when it can optimize out function calls. This function uses `stable`:

- `stable` used for functions that return the same value for the same input params (most common)
- `volatile` used for functions that can return different values for the same input param (e.g. random)
- `immutable` used for functions that always return constants

And the Python function body is between the `$$`, followed by the language definition.

Library Support

Redshift's UDFs have access to the full standard library and several popular packages including `numpy`, `pandas`, `python-dateutil`, `pytz`, and `scipy`. Here's an example of importing the JSON library to make working with JSON arrays easy:

```
/*
JSON_ARRAY_SORT
Returns sorts a JSON array and returns it as a string,
  second param sets direction
Examples:
  select json_array_sort('["a","c","b"]', true)
    --> '["a", "b", "c"]'
  select json_array_sort('[1, 3, 2]', true)
    --> '[1, 2, 3]'
*/
create or replace function json_array_sort
    (j varchar(max), ascending boolean)
  returns varchar(max)
  stable as $$
    import json
    if not j:
      return None
```

```
  try:
    arr = json.loads(j)
  except ValueError:
    return None
  if not ascending:
    arr = sorted(arr, reverse=True)
  else:
    arr = sorted(arr)
  return json.dumps(arr)
$$ language plpythonu;
```

Keep in mind that UDFs are per-database. So if you have multiple databases in your Redshift cluster, you'll need to add the UDFs to each database.

This week, Periscope's data cache will be updated with all of these UDFs. If you sign up for Periscope, you'll have access to all of these functions right away!