Amazon Redshift Scalar User-Defined Functions: A How-To

Scott Hoover, *Data Scientist*
Sep 11, 2015

https://looker.com/blog/amazon-redshift-user-defined-functions

Overview

Amazon Web Services (AWS) recently announced that Redshift, their managed MPP database offering, will support scalar user-defined functions (hereinafter UDFs). Redshift's UDFs rely on Python to define a program that transforms input data. Because Python is such an accessible language with a host of libraries, many users will find writing UDFs in Redshift more approachable than other SQL dialects, which tend to rely on low-level, compiled programming languages for UDF construction.

This article is intended to give the technical user insight into how to create, use, and troubleshoot scalar UDFs in Redshift, as well as provide a few recipes to get started.

Scalar Functions

A scalar function returns a single result value for each input value. Performed over *N* rows, we'd expect our scalar function to return *N* values. There are five basic elements required to implement a scalar UDF in Redshift:

- first, one must specify a **function name**, which, along with its input arguments and input data types, makes it unique from any other native or user-defined functions (AWS recommends an `f_` prefix in the name);
- second, one must specify the function's **arguments and data types**, which reference some sort of named argument like "date_column" or "zipcode" and the corresponding Redshift data type—*e.g.*, `date_column TIMESTAMP` or `zipcode VARCHAR`;
- third, one must specify the function's output or **return data type**, which tends to be, but is not necessarily, the same as the input data type;
- fourth, one must specify the function's **volatility**, which informs the optimizer about how the function will behave given repeated function calls against the same input data;
- fifth, one must supply a **Python program** to perform some transformation to the input data.

See the official AWS documentation for specifics regarding function volatility as well as data-type mappings between Redshift and Python. This is the generalized syntax to create a scalar UDF.

```
CREATE [OR REPLACE] FUNCTION my_udf ( [argument redshift_data_type,
...] )
    RETURNS redshift_data_type
{ STABLE | VOLATILE | IMMUTABLE }
AS $$
    my_python_program
$$ LANGUAGE plpythonu;
```

The Python program is placed between the set of `$$`; the language, for the time being, is always `plpythonu`. Let's walk through an example of a scalar function that, for any input timestamp, extracts the week starting Sunday in `%Y-%m-%d` format. (For simplicity, I'm going to assume the input is a timestamp without time zone.)

```
CREATE FUNCTION week_starting_sunday (date_col TIMESTAMP)
    RETURNS VARCHAR
STABLE
AS $$
    from datetime import datetime, timedelta
    day = str(date_col)
    dt = datetime.strptime(day, '%Y-%m-%d %H:%M:%S')
    week = dt - timedelta((dt.weekday() + 1) % 7)
    return week.strftime('%Y-%m-%d')
$$ LANGUAGE plpythonu;
```

We can run a simple test on this function using

```
SELECT CURRENT_DATE AS today
  , week_starting_sunday(CURRENT_DATE);
```

which returns

today	**week_starting_sunday**
2015-03-26	2015-03-22

We can also write a scalar function that takes multiple columns as input. Take, for instance, a situation where we have latitude and longitude for both origin and destination. Let's write a UDF, called `DISTANCE`, that uses the haversine formula to calculate approximate distance in miles between origin and destination.

```
CREATE FUNCTION DISTANCE (orig_lat float, orig_long float, dest_lat
float, dest_long float)
  RETURNS float
STABLE
```

```
AS $$
  import math
  r = 3963.1676          # earth's radius, in miles
  phi_orig = math.radians(orig_lat)
  phi_dest = math.radians(dest_lat)
  delta_lat = math.radians(dest_lat - orig_lat)
  delta_long = math.radians(dest_long - orig_long)
  a = math.sin(delta_lat/2) * math.sin(delta_lat/2) +
math.cos(phi_orig) \
      * math.cos(phi_dest) * math.sin(delta_long/2) *
math.sin(delta_long/2)
  c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
  d = r * c
  return d
$$ LANGUAGE plpythonu;
```

Let's test this function on a table that captures flight information.

id	origin_latitude	origin_longitude	destination_latitude	de
1	37.775	-122.4183333	40.64377899999999	-73.782
2	40.64377899999999	-73.78227400000003	37.775	-122.41
3	37.775	-122.4183333	42.3656132	-71.009

Running our UDF

```
SELECT id
  , DISTANCE(origin_latitude, origin_longitude, destination_latitude,
destination_longitude) AS distance_in_miles
FROM flights
```

yields

id	distance_in_miles
1	2580.96027955641
2	2580.96027955641

3	2698.1617901510726

Closing Thoughts

Scalar UDFs can be quite useful in certain circumstances. If performance is comparable, a UDF may be ideal to replace a difficult-to-follow stored procedure or go-to SQL transformation. This can make grasping the objective of an S-Q-L query more manageable. There's also an opportunity to take slow-running procedures and speed them up with an imperative program written in a language that makes transforming data much simpler.

The choice to go with Python for UDF construction certainly opens up some doors. Data scientists should be able to move some interesting data exploration techniques typically reserved for Python (perhaps ran on sample data or using the Map-Reduce framework) back into SQL to be run on the full data set with less tedium.