Pivotal

A NEW PLATFORM FOR A NEW ERA

User Defined Functions

Pivotal® Greenplum Database

Pivotal™

Agenda

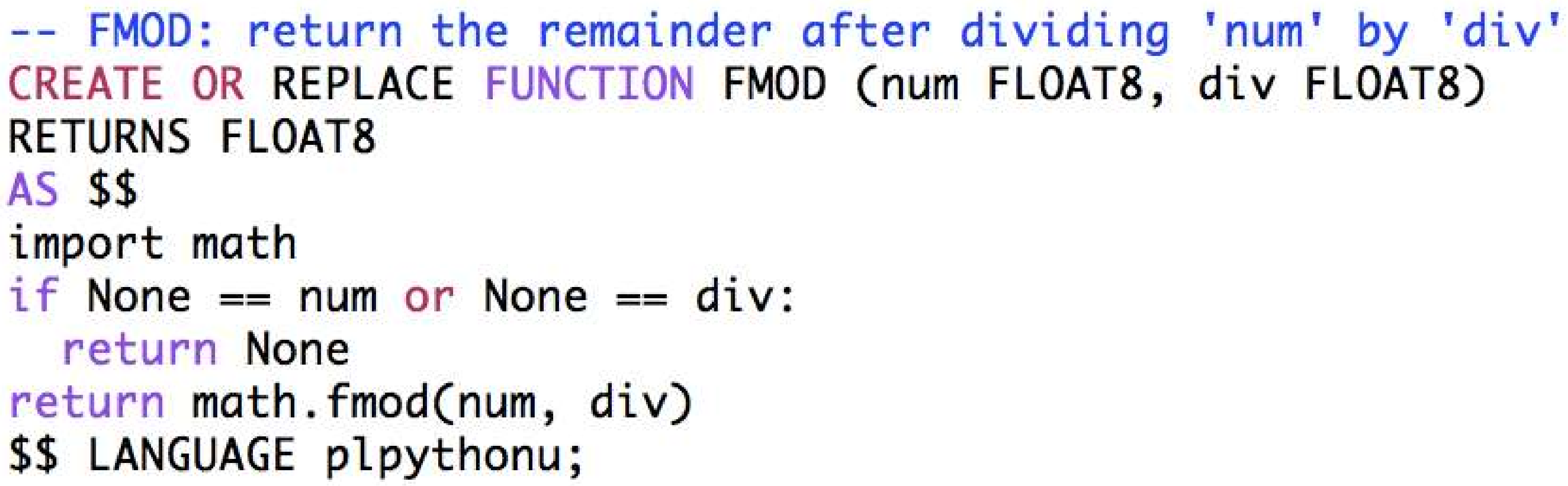
- User Defined Functions
  - Why use them
  - How to create them
  - Some important things you need to know
  - How to use them

User Defined Functions -- UDF

- PostgreSQL and Greenplum do not have stored procedures, but functions.
- Common tasks can be made easier to maintain by keeping the code in a function.
- Allows data access in a restricted manner.
- Allows users to access and manipulate data without knowing the implementation details or having to code.
- Madlib creates UDFs that allows users access to machine learning tools without their knowing how to code them.

Pivotal™

Example: Python Floating Point "modulus"



```
-- FMOD: return the remainder after dividing 'num' by 'div'
CREATE OR REPLACE FUNCTION FMOD (num FLOAT8, div FLOAT8)
RETURNS FLOAT8
AS $$
import math
if None == num or None == div:
  return None
return math.fmod(num, div)
$$ LANGUAGE plpythonu;
```

Types of User Defined Functions

Greenplum supports several function types, including:

- Query language functions where the functions are written in SQL
- Procedural language functions where the functions are written in:
  - PL/pgSQL
  - Perl
  - Python
  - R
  - Java
- Internal functions
- C-language functions

Note: Greenplum supports PL/pgSQL, PL/Perl, and PL/Python out of the box. Other extensions can be added with the `gppkg` utility and registered with the `createlang` utility.

Pivotal™

Function Volatility Categories

Functions belong to one of the following categories:

- `IMMUTABLE` – Functions rely only on information passed in and always returns the same values given the same argument values
- `STABLE` – Functions:
  - Consistently return the same result for the same argument values in a single table scan
  - Can return different values across SQL statements
- `VOLATILE` – Functions in this category:
  - Are evaluated on each row
  - Can return different values within a single table scan

Note: `IMMUTABLE` and `VOLATILE` functions are executed on the master and not at the segment level.

Pivotal™

Security Invoker or Definer

- When you execute a function written by another to which you have execute privilege, do you obey the acess privileges of the definer of the function or the user or invoker of the function.

- SECURITY INVOKER (the default) indicates that the function is to be executed with the privileges of the user that calls it.
- SECURITY DEFINER specifies that the function is to be executed with the privileges of the user that created it.

Creating, Modifying, and Dropping Functions

Action	SQL Syntax
Create a function	`CREATE FUNCTION`
Replace a function	`CREATE OR REPLACE FUNCTION`
Change a function	`ALTER FUNCTION`
Drop or remove a function	`DROP FUNCTION`

There are several things to note when managing functions:

- Functions operating on tables must be created in the same schema as the table
- Removing a function requires that the `DROP` statement include the parameter types. For example:
  ```
  DROP FUNCTION DIMENSIONS.getcust(int);
  ```

Query Language Function – Rules

An SQL function:

- Executes an arbitrary set of SQL commands
- Returns the results of the last query in the list
- Returns the first row of the result set of the last statement executed
- Can return a set of a previously defined data type
- Must end with a semicolon at the end of each statement in the body of the function

SQL functions:

- May contain `SELECT` statements
- May contain DML statements, such as `INSERT`, `UPDATE`, or `DELETE` statements
- May not contain `ROLLBACK`, `SAVEPOINT`, `BEGIN`, or `COMMIT` commands
- Must use `SELECT` in the last line unless the return type is void.

Query Language Function Structure

When developing SQL functions:

- Use doubled single quote syntax to reference quoted values in the query
- You can pass in zero or more parameters to a function. Parameters are:
  - A sequential list referenced by $*n* notation
  - Start at $1 for the first parameter and increase for each additional parameter

Function Overloading

Function overloading:

- Lets you define multiple functions with the same name
- Must have different input parameter types or number
- Allows the appropriate version of the function to be called by the optimizer based on the input data type and number of arguments

```
CREATE FUNCTION get_customer_ids(int)
    …
LANGUAGE SQL;
```

≠

```
CREATE FUNCTION get_customer_ids(char)
    …
LANGUAGE SQL;
```

Function Overload – Example

```
CREATE FUNCTION dimensions.getcust(int)
        RETURNS customer AS $$
SELECT *
FROM dimensions.customer WHERE customerid = $1;
$$ LANGUAGE SQL;
```

This function accepts an INTEGER parameter

```
CREATE FUNCTION dimensions.getcust(char)
        RETURNS customer AS $$
SELECT * FROM dimensions.customer WHERE state = $1;
$$ LANGUAGE SQL;
select *,upper(custname)from dimensions.getcust('WA') ;
select *,upper(custname) from dimensions.getcust(1);
```

This function accepts a CHAR parameter

Pivotal

Query Language Function – Example

The following example shows a function that has no parameters or return set:

```
CREATE FUNCTION public.clean_customer()
  RETURNS void AS ‘DELETE FROM dimensions.customer
    WHERE state = ‘‘NA’’; ’
 LANGUAGE SQL;
```

The function is called as follows:

```
SELECT public.clean_customer();

clean_customer
-----------
(1 row)
```

Query Language Function – With Parameter

The following example shows a function that has no parameters or return set:

```
CREATE FUNCTION public.clean_specific_customer
(which char(2))
  RETURNS void AS ‘DELETE FROM dimensions.customer
    WHERE state = $1; ’
 LANGUAGE SQL;
```

The function is called as follows:

```
SELECT public.clean_specific_customer(‘NA’);

clean_specific_customer
-----------------------


(1 row)
```

SQL Functions on Base Types

In this example, a single value is returned to the caller:

```
CREATE FUNCTION public.customer_cnt(char)
  RETURNS bigint AS ‘SELECT COUNT(*)
                                FROM dimensions.customer
    WHERE state = $1; ’
 LANGUAGE SQL;
```

The function is called as follows:

```
SELECT public.customer_cnt(‘WA’);

customer_cnt
-----------
176
```

SQL Functions on Composite Types

The following example shows how to pass a composite parameter, in this case, a table:

```
CREATE FUNCTION facts.viewnewtaxamt(sale) RETURNS numeric AS
$$
   SELECT $1.taxamt * .90;
$$ LANGUAGE SQL;
```

Start function body

Function body

End function body

The function is called as follows:

```
SELECT transid, transdate, taxamt,
facts.viewnewtaxamt(txn_id)
   FROM transaction;
```

SQL Functions with Output Parameters

The following function does not call the `RETURNS` clause, instead passing a value out of the function using an `OUT` parameter:

```
CREATE FUNCTION public.cust_cnt(IN whichstate char ,
                                OUT MyCount bigint)
AS ‘SELECT COUNT(*)
      FROM dimensions.customer
    WHERE state = $1; ’
 LANGUAGE SQL;
```

The function is called as follows:

```
SELECT public.cust_cnt(‘WA’);
customer_cnt
-----------
176
```

SQL Functions as Table Sources

The following shows a function that accepts an integer and returns a composite parameter:

```
CREATE FUNCTION dimensions.getcust(int)
       RETURNS customer AS $$
   SELECT *
     FROM dimensions.customer WHERE customerid = $1;
$$ LANGUAGE SQL;
```

The function is called as follows:

```
SELECT *,UPPER(custname)
   FROM dimensions.getcust(100);
```

SQL Functions Returning Sets

The following example returns all rows to the caller using `SETOF` instead of returning only the first row:

```
CREATE FUNCTION dimensions.getstatecust(char)
       RETURNS SETOF customer AS $$
   SELECT *
     FROM dimensions.customer WHERE state = $1;
$$ LANGUAGE SQL;
```

The function is called as follows:

```
SELECT *,UPPER(city)
FROM dimensions.getstatecust('WA');
```

Procedural Language Functions

PL/pgSQL procedural language for Greenplum:

- Can be used to create functions
- Adds control structures to the SQL language
- Can perform complex computations
- Inherits all user-defined types, functions, and operators
- Can be defined to be trusted by the server
- Is *relatively* easy to use

Trusted and Untrusted Languages

- A trusted language is one that is executed entirely within the database server, with NO access to operating system commands.
- An untrusted language is one which is run as the GPDB super user and has the ability to use operating system calls to do whatever gpadmin can do, e.g. read all the database files.
- Only gpadmin or roles with gpadmin privileges can create functions in untrusted language.
- Trusted languages: SQL, pgPLSQL, and plperl
- Untrusted languages: plr, plpython,c
- Consult pg_catalog.pg_language to see which languages are installed

How to create functions in untrusted languages for general use

- Non gpadmin users test the function on a non production system or single node sandbox.
- When satisified with the function, the user passes the code to a user with gpadmin privileges on the production system.
- The privileged user creates the function and gives execution privileges on the production system.
- Should the function have definer rights or invoker rights? It depends upon the use case.

References

- All the Pivotal Greenplum Database documentation is available online
  - Reference Guide
  - Admin Guide
  - Client Tools
  - Load Tools
  - Connectivity Tools
  - Many more
- Pivotal Greenplum Database documentation:
  - http://gpdb.docs.pivotal.io/gpdb-438.html

Thank You

Pivotal

A NEW PLATFORM FOR A NEW ERA