Multicorn Documentation

Release 1.1.1

Ronan Dunklau, Florian Mounier

Jul 14, 2017

Contents

Multicorn is a PostgreSQL 9.1+ extension meant to make Foreign Data Wrapper development easy, by allowing the programmer to use the Python programming language.

If you just wanto use it as soon as possible, jump straight to the *installation* section.

Contents:

CHAPTER 1

Installation

Requirements

- Postgresql 9.1+
- Postgresql development packages
- Python development packages
- python 2.6 or >= python 3.3 as your default python

If you are using *PostgreSQL 9.1*, you should use the 0.9.1 release.

If you are using *PostgreSQL 9.2* or superior, you should use the 1.0.0 series. (Currently 1.0.1).

If you are using Debian, a packaging effort is ongoing for PostgreSQL 9.4. You can install it from here.

With the pgxn client:

```
pgxn install multicorn
```

From pgxn:

```
wget http://api.pgxn.org/dist/multicorn/1.1.1/multicorn-1.1.1.zip
unzip multicorn-1.1.1
cd multicorn-1.1.1
make && sudo make install
```

From source:

```
git clone git://github.com/Kozea/Multicorn.git
cd Multicorn
make && make install
```

CHAPTER 2

Usage

The multicorn foreign data wrapper is not different from other foreign data wrappers.

To use it, you have to:

- Create the extension in the target database. As a PostgreSQL super user, run the following SQL:

```
CREATE EXTENSION multicorn;
```

- Create a server. In the SQL `OPTIONS` clause, you must provide an options named wrapper, containing the fully-qualified class name of the concrete python foreign data wrapper you wish to use. want to use:

```
CREATE SERVER multicorn_imap FOREIGN DATA WRAPPER multicorn
options (
  wrapper 'multicorn.imapfdw.ImapFdw'
);
```

You can then proceed on with the actual foreign tables creation, and pass them the needed options.

Each foreign data wrapper supports its own set of options, and may interpret the columns definitions differently.

You should look at the documentation for the specific *Foreign Data Wraper documentation*

CHAPTER 3

Included Foreign Data Wrappers

Multicorn is bundled with a small set of Foreign Data Wrappers, which you can use or customize for your needs.

SQLAlchemy Foreign Data Wrapper

LDAP Foreign Data Wrapper

Purpose

This fdw can be used to access directory servers via the LDAP protocol. Tested with OpenLDAP. It supports: simple bind, multiple scopes (subtree, base, etc)

Dependencies

If using Multicorn >= 1.1.0, you will need the ldap3 library:

For prior version, you will need the ldap library:

Required options

`uri` (string) The URI for the server, for example "ldap://localhost".

`path` (string) The base in which the search is performed, for example "dc=example,dc=com".

`objectclass` (string) The objectClass for which is searched, for example "inetOrgPerson".

`scope` (string) The scope: one, sub or base.

Optional options

`binddn` (string) The binddn for example 'cn=admin,dc=example,dc=com'.

`bindpwd` (string) The credentials for the binddn.

Usage Example

To search for a person definition:

```
CREATE SERVER ldap_srv foreign data wrapper multicorn options (
    wrapper 'multicorn.ldapfdw.LdapFdw'
);

CREATE FOREIGN TABLE ldapexample (
    mail character varying,
    cn character varying,
    description character varying
) server ldap_srv options (
    uri 'ldap://localhost',
    path 'dc=lab,dc=example,dc=com',
    scope 'sub',
    binddn 'cn=Admin,dc=example,dc=com',
    bindpwd 'admin',
    objectClass '*'
);

select * from ldapexample;
```

```
         mail           |        cn       |    description
------------------------+-----------------+--------------------
 test@example.com       | test            |
 admin@example.com      | admin           | LDAP administrator
 someuser@example.com   | Some Test User  |
(3 rows)
```

Process Foreign Data Wrapper

CHAPTER 4

Writing an FDW

If you want to write an FDW, we recommend you start with the *Tutorial: Writing an FDW*.

Tutorial: Writing an FDW

Multicorn provides a simple interface for writing foreign data wrappers: the `multicorn.ForeignDataWrapper` interface.

Implementing a foreign data wrapper is as simple as inheriting from `multicorn.ForeignDataWrapper` and implemening the `execute` method.

What are we trying to achieve ?

Supposing we want to implement a foreign data wrapper which only returns a set of 20 rows, containing in each column the name of the column itself concatenated with the number of the line.

The goal of this tutorial is to be able to execute this:

```
CREATE FOREIGN TABLE constanttable (
    test character varying,
    test2 character varying
) server multicorn_srv options (
    wrapper 'myfdw.ConstantForeignDataWrapper'
)

SELECT * from constanttable;
```

And obtain this as a result:

```
  test    |  test2
---------+----------
 test 0   | test2 0
 test 1   | test2 1
 test 2  | test2 2
 test 3  | test2 3
 test 4  | test2 4
 test 5  | test2 5
 test 6  | test2 6
 test 7  | test2 7
 test 8  | test2 8
 test 9  | test2 9
 test 10 | test2 10
 test 11 | test2 11
 test 12 | test2 12
 test 13 | test2 13
 test 14 | test2 14
 test 15 | test2 15
 test 16 | test2 16
 test 17 | test2 17
 test 18 | test2 18
 test 19 | test2 19
(20 lignes)
```

How do we do that ?

The fdw described above is pretty simple, implementing it should be easy !

First things first, we have to create a new python module.

This can be achieved with the most simple `setup.py` file:

```
import subprocess
from setuptools import setup, find_packages, Extension

setup(
  name='myfdw',
  version='0.0.1',
  author='Ronan Dunklau',
  license='Postgresql',
  packages=['myfdw']
)
```

But let's see the whole code. To be usable with the above `CREATE FOREIGN TABLE` statement, this module should be named `myfdw`.

```
from multicorn import ForeignDataWrapper

class ConstantForeignDataWrapper(ForeignDataWrapper):

    def __init__(self, options, columns):
        super(ConstantForeignDataWrapper, self).__init__(options, columns)
        self.columns = columns

    def execute(self, quals, columns):
        for index in range(20):
            line = {}
            for column_name in self.columns:
                line[column_name] = '%s %s' % (column_name, index)
            yield line
```

You should have the following directory structure:

```
.
|-- myfdw/
|   `-- __init__.py
`-- setup.py
```

To install it, just run `python setup.py install`, and the file will be copied to your global python installation, which should be the one your PostgreSQL instance is using.

And that's it ! You just created your first foreign data wrapper. But let's look a bit more thoroughly to the class...

The first thing to do (although optional, since you can implement the interface via duck-typing), is to import the base class and subclass it:

```
from multicorn import ForeignDataWrapper

class ConstantForeignDataWrapper(ForeignDataWrapper):
```

The init method must accept two arguments

`options` A dictionary of options given in the `OPTIONS` clause of the `CREATE FOREIGN TABLE` statement, minus the wrapper option.

`columns` A mapping of the columns names given during the table creation, associated to their types. Ex: {'test': 'character varying'}

Our access point do not need any options, thus we will only need to keep a reference to the columns:

```
def __init__(self, options, columns):
    super(ConstantForeignDataWrapper, self).__init__(options, columns)
    self.columns = columns
```

The execute method is the core of the API. It is called with a list of `Qual` objects, and a list column names, which we will ignore for now but more on that *later*.

This method must return an iterable of the resulting lines. Each line can be either a list containing an item by column, or a dictonary mappning the column names to their value.

For this example, we chose to build a dictionary. Each column contains the concatenation of the column name and the line index.

```
def execute(self, quals):
    for index in range(20):
        line = {}
        for column_name in self.columns:
            line[column_name] = '%s %s' % (column_name, index)
        yield line
```

And that's it !

Write API

Since PostgreSQL 9.3, foreign data wrappers can implement a write API.

In multicorn, this involves defining which column will be used as a primary key (mandatory) and implementing the following methods at your discretion:

```
def insert(self, new_values)
def update(self, old_values, new_values)
def delete(self, old_values)
```

Each of these arguments will be dictionaries, containing at least the column you defined as a primary key, and the values to insert or those which have changed (for an update). In addition, other values may be present depending on the query involved.

These methods should return a dictionary containing the new values (after insertion or update). This will be used in the case of RETURNING clauses of the form:

```
INSERT INTO my_ft VALUES (some_value) RETURNING *;
```

You can return new values if the values that were given in sql are not the ones that are actually stored (think about default values, triggers...).

The row_id_column attribute must be set to the name of a column acting as a primary key. For example:

```
class MyFDW(ForeignDataWrapper):

  def __init__(self, fdw_options, fdw_columns):
    self.row_id_column = fdw_columns.keys()[0]
```

If you want to handle transaction hooks, you can implement the following methods:

```
def commit(self)
def rollback(self)
def pre_commit(self)
```

The pre_commit method will be called just before the local transaction commits. You can raise an exception here to abort the current transaction were your remote commit to fail.

The commit method will be called just at commit time, while the rollback method will be called whenever the local transaction is rollbacked.

Optimizations

As was noted in the code commentaries, the execute methods accept a `quals` argument. This argument is a list of quals object, which are defined in multicorn/__init__.py. A Qual object defines a simple condition wich can be used by the foreign data wrapper to restrict the number of the results. The Qual class defines three instance's attributes:

- field_name: the name of the column concerned by the condition.
- operator: the name of the operator.
- value: the value expressed in the condition.

Let's suppose we write the following query:

```
SELECT * from constanttable where test = 'test 2' and test2 like '%3%';
```

The method execute would be called with the following quals:

```
[Qual('test', '=', 'test 2'), Qual('test', '~~', '3')]
```

Now you can use this information to reduce the set of results to return to the postgresql server.

Note: You don't HAVE to enforce those quals, Postgresql will check them anyway. It's nonetheless useful to reduce the amount of results you fetch over the network, for example.

Similarly, the columns argument contains the list of needed columns. You can use this information to reduce the amount of data that has to be fetched.

For example, the following query:

```
select test, test2 from constanttable;
```

would result in the following columns argument:

```
['test', 'test2']
```

Once again, if you returns more than these columns everything should be fine.

Parameterized paths

The python FDW implementor can affect the planner by implementing the get_path_keys and get_rel_size methods.

```
def get_rel_size(self, quals, columns):
```

This method must return a tuple of the form (expected_number_of_row, expected_mean_width_of_a_row (in bytes)).

The quals and columns arguments can be used to compute those estimates.

For example, the imapfdw computes a huge width whenever the payload column is requested.

```
def get_path_keys(self):
```

This method must return a list of tuple of the form (column_name, expected_number_of_row).

The expected_number_of_row must be computed as if a "where column_name = some_value" filter were applied.

This helps the planner to estimate parameterized paths cost, and change the plan accordingly.

For example, informing the planner that a filter on a column may return exactly one row, instead of the full billion, may help it on deciding to use a nested-loop instead of a full sequential scan.

Error reporting

In the multicorn.utils module lies a simple utility function, `log_to_postgres`.

This function is mapped to the Postgresql function erreport.

It accepts three arguments:

`message` (required) A python string containing the message to report.

`level` (optional, defaults to `logging.INFO`)

The severity of the message. The following values are accepted:

- **`logging.DEBUG`** Maps to a postgresql DEBUG1 message. In most configurations, it won't show at all.
- **`logging.INFO`** Maps to a postgresql NOTICE message. A NOTICE message is passed to the client, as well as in the server logs.

`logging.WARNING` Maps to a postgresql WARNING message. A WARNING message is passed to the client, as well as in the server logs.

`logging.ERROR` Maps to a postgresql ERROR message. An ERROR message is passed to the client, as well as in the server logs.

Important: An ERROR message results in the current transaction being aborted. Think about the consequences when you use it !

`logging.CRITICAL` Maps to a postgresql FATAL message. Causes the current server process to abort.

Important: A CRITICAL message results in the current server process to be aborted Think about the consequences when you use it !

`hint` (optional) An hint given to the user to resolve the cause of the message (ex:`Try adding the missing option in the table creation statement`)

Foreign Data Wrapper lifecycle

The foreign data wrapper associated to a table is instantiated on a per-process basis, and it happens when the first query is run against it.

Usually, postgresql server processes are spawned on a per-connection basis.

During the life time of a server process, the instance is cached. That means that if you have to keep references to resources such as connections, you should establish them in the `__init__` method and cache them as instance attributes.

CHAPTER 5

Multicorn Internal Design

This part is more geared toward those who may want to hack on Multicorn itself.

PostgreSQL C API

The PostgreSQL C API follows a pretty simple workflow.

CHAPTER 6

Contribute

Send Us an Mail

Want to write kind words? You can send a mail on our Librelist mailing-list <mailto:multicorn@librelist.com> and even take a look at .. meta: mailarchives_‘the archives‘.

If you use Multicorn in production, we would love to hear about your use-case !

Report Bugs

Found a bug? Want a new feature? Report a new issue on the *Multicorn bug-tracker on GitHub <https://github.com/Kozea/Multicorn/issues/>*.

Hack

Interested in hacking? Feel free to clone the *git repository on GitHub <https://github.com/Kozea/Multicorn>* if you want to add new features, fix bugs or update documentation.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index