Don't Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes

Pedro Holanda[1], **Mark Raasveldt**[1], **Martin Kersten**[1]

[1] Centrum Wiskunde & Informatica (CWI)
Amsterdam, The Netherlands

{holanda,m.raasveldt,mk}@cwi.nl

Abstract. *User-defined functions (UDFs) are an integral part of performing in-database analytics. Executing data analysis inside a database provides significant improvements over traditional methods, such as close-to-the-data execution, low conversion overhead and automatic parallelization. However, UDFs have poor support for debugging. Since they are executed from within the database process, traditional debugging tools such as Integrated Development Environments (IDEs) and Read-Eval-Print Loops (REPLs) cannot be used during development. As a result, writing functional UDFs is challenging. In this paper, we present an extension to the open-source database system MonetDB that allows developers to debug their UDFs using modern debugging techniques.*

1. Introduction

Data scientists rely on scripting languages, such as R and Python, to perform data analysis tasks. Combining these languages with a database has many advantages. The traditional method of using a database in conjunction with these scripting languages is to connect to a database using a client protocol. The data is then transferred from the database to the analytical tool. However, this is very inefficient when dealing with large amounts of data [Raasveldt and Mühleisen 2017].

In-database analytics promises to solve this issue. By performing the analytics inside the database, the data transfer overhead is mitigated [Raasveldt and Mühleisen 2016]. The primary way of performing in-database analytics is through the use of UDFs. As shown in Table 1, most database vendors support UDFs in at least one scripting language frequently used for analysis.

While UDFs can perform highly efficient data analysis, debugging them has been a long-standing problem. Because the UDFs are executed within the database server, developers cannot use sophisticated debugging techniques (e.g., Interactive Debugging). Instead, they have to resort to inefficient debugging strategies in an effort to make their code work.

With the increasing number of R and Python packages, these analytical programs are quickly growing in complexity and scope. Finding and correcting bugs in complex programs is a challenging and time consuming task. In some cases this can take up to 50% of development time [Hailpern and Santhanam 2002]. However, when using more primitive debugging techniques, such as print or log-file debugging, this can take even longer.

Table 1. DBMSs Support for UDFs

DBMS	Python	R
Greenplum	X	X
DB2	-	-
MonetDB	X	X
MySQL	-	-
Oracle	-	X
PostgreSQL	X	X
Redshift	X	-
SQL Server	X	X
SQLite	X	-
Vertica	-	X

In this paper, we propose a solution to this problem. From within a client connected to a database, we ask the user to provide us with a problematic SQL statement that they want to debug. We then analyze it, and extract any required input data from the database server. We transfer the input data and the source code of the UDF to the client. Finally, we execute the function in the client process exactly as it would have been executed in the database server. The developer can then easily use interactive debugging tools to aid him in creating and modifying the UDFs.

The main contributions of this paper are:

- We introduce a novel strategy that allows developers to use interactive debugging techniques in conjunction with UDFs.
- We provide an Open-Source implementation of our work for the popular database MonetDB.
- We show our debugging framework being used for a concrete use-case, and compare it against the old ways of debugging UDFs.

Outline. This paper is organized as follows. Section 2 discusses the main debugging techniques and the state-of-the-art debugging for scripting UDFs. In Section 3 we describe our solution and demonstrate a concrete use-case that shows the difficulties of Python UDF debugging. In Section 4 we present our conclusions and discuss future work.

2. Background & Related Work

Debugging is the process of localizing, classifying, understanding and repairing an error in a piece of software. Many debugging techniques have been developed and are frequently used in the software development process [McConnell 2004]. The most commonly used debugging techniques are:

- **Print Debugging.** The developer adds print statements to the code in order to track the values of different variables while the code is running. This technique has several disadvantages. The developer needs to decide upfront which variables he wants to track and where he wants to track them. If the developer later decides that different variables should be tracked, he needs to recompile and rerun the code. Another disadvantage is that temporary code has to be inserted into the software, that then need to be removed after fixing the bug. Print debugging also works poorly in multithreaded programs, because print statements from different threads get mixed in the output stream.

- **Log-File Debugging.** Log-File debugging is similar to print debugging, however, instead of writing to a standard output stream the developer writes output to a log-file. This has an advantage over print-debugging, in that log files are kept around even after the console has been closed. However, it still suffers from many of the same problems as print-debugging, as it requires the source code to be altered by the programmer upfront.
- **Interactive Debugging.** Interactive debugging allows the developer to inspect the state of a program while it is running. It allows the developer to work interactively with the source code, printing any variables he thinks are relevant to the program, and even stepping through the code line-by-line. Certain interactive debuggers even allow the developer to change the source code while the program is running. This form of debugging can be used without any modification to the source code. The developer only needs to attach a debugger to a running program to start his debugging session.

Scripting languages have several popular interactive debugging tools. The two main types of tools are console debugging (i.e. REPLs) and IDEs. Python has many different options for both REPLs and IDEs, such as pdb[1], IPython[Pérez and Granger 2007], PyDev[2] and PyCharm[3]. However, these cannot be used together with UDFs because these debugging tools need to control the code execution. This is not possible for code that runs within a database process, since the database controls the code flow. Consequently, developers of UDFs are forced to use either print or log-file debugging.

3. Debugging UDFs

The main issue with using interactive debuggers in conjunction with UDFs is that the execution of the UDFs happens inside the database server. To control the execution of the UDF, the debugger needs to be in control of the code. Hence, client-local execution of UDFs is necessary.

To accomplish client-local execution we transfer the UDF from the database server to the clients' machine and execute it in the local environment. A challenge here is that errors can be data-dependent, and the input data of the UDFs depend on the query that the UDF is used in. Consequently developers of UDFs often want to debug their UDFs in the context of a specific SQL query. To facilitate this, we allow developers to input an arbitrary SQL query that involves a call to the UDF they want to inspect. Both the UDF and the required input data are then transferred from the database and sent to the client. Afterwards they are either exported to a file for use in external IDEs, or directly executed in an interactive debugger.

The problem with transferring the input data of UDFs from the database to the client is that the amount of data can be very large. This is only required for debugging, however. The actual execution of the UDFs still occurs completely inside the database server. Still, it is possible that the clients' machine does not have space for all the required

[1]`https://docs.python.org/2/library/pdb.html`
[2]`www.pydev.org`
[3]`www.jetbrains.com/pycharm`

input data. For this reason, we also introduce an optional sampling step that only transfers a uniform random sample of the data instead of the full input data set.

Listing 1 depicts an example of a UDF that is supposed to compute the mean absolute deviation of a given column using the following formula:

$$\text{mean\_deviation}(X) = \frac{\sum |X - X'|}{N}$$

where X represents a column, X' the columns' mean value and N the columns' size. However, the example has a bug. On line 9, the regular difference is calculated instead of the absolute difference, producing a semantic error (i.e., syntactically correct but logically incorrect). This bug can be fixed by computing the absolute difference instead. However, to locate the bug, the developer must inspect the values of variables throughout the programs' execution.

Listing 1. Mean Deviation UDF

```
 1  CREATE FUNCTION mean_deviation(column INTEGER)
 2  RETURNS DOUBLE LANGUAGE PYTHON {
 3      mean = 0
 4      for i in range (0, len(column)):
 5          mean += column[i]
 6      mean = mean / len(column)
 7      distance = 0
 8      for i in range (0, len(column)):
 9          distance += column[i] - mean
10      deviation = distance/len(column)
11      return deviation;
12  };
```

To use print debugging to locate the error, the developer must alter the original source code. The source code must be modified to include print statements that track variable changes. Using Listing 1 as an example, print statements could be introduced after line 6 and 10 to track the mean and distance variables respectively. This solution requires many iterations, placing the print statements in different positions and tracking different variables. Each iteration includes a modification to the source code and requires rerunning of the function, which can be very time consuming. Finally, after finding and fixing the bug, the print statements must be removed.

We can see that print debugging is very cumbersome, and requires rerunning the query multiple times before the bug is found, even in this simple example. Listing 2 depicts how the developer can debug this function using our solution. First, the developer connects to a running database using the Python client (pymonetdb[4] for MonetDB). Then, the developer creates a SQL query involving the UDF, just as he would if he were to normally execute the query. However, instead of executing it, he calls either the `debug` function (line 5) for console debugging or the `export` function (line 6) for IDE debugging.

[4]https://pymonetdb.readthedocs.io/

Listing 2. Debugging UDF through pymonetdb

```
import pymonetdb
conn = pymonetdb.connect(database='demo')
c = conn.cursor()
sql = 'select mean_deviation(id) from tables;'
c.debug(sql, 'mean_deviation')
c.export(sql, 'mean_deviation')
```

The `debug` function directly executes the UDF inside the client environment and attaches an interactive console debugger (pdb) to the UDF. The developer can then use the debugger to step through the code, and obtain the values of variables throughout the codes' execution. In addition, the developer can set breakpoints or watchpoints to automatically monitor changes to variables.

The `export` function writes the UDF and the input data to a file, which can then be inspected by the developer and imported into an IDE of his choice. The IDE can then be used to interactively debug the function and step through the code.

Both functions take an optional sample parameter, that can be used to control the maximum allowed input size. This way, transfer of a large amount of data to the client can be avoided.

Loopback Query UDF

MonetDB/Python supports loopback queries inside UDFs. Loopback queries allow users to query the database directly from within the UDF. The results of the query are converted to the host language of the UDFs. In Python UDFs, they can be can used through the `_conn` object that is passed to every UDF. Loopback queries are useful because they can bypass cardinality restrictions of the relational querying model. Listing 3 depicts an example of a UDF that uses a loopback query to retrieve a classifier from the database, and subsequently uses the classifier on its input data.

Listing 3. Loopback Queries

```
CREATE FUNCTION classify(id INTEGER, value INTEGER)
RETURNS TABLE(id INTEGER, prediction STRING)
LANGUAGE PYTHON
{
    import pickle
    res = _conn.execute("SELECT * FROM classifier WHERE name='RFC';")
    classifier = pickle.loads(res['classifier'][0])
    return {'id': id, 'prediction': classifier.predict(value) }
};
```

Loopback queries are a challenge for client-local execution. Because the UDF no longer runs inside the database, it cannot be queried in the same fashion, hence they have to be modified to work in this scenario. One solution would be to run all the loopback queries when exporting the function and retrieving their results in addition to the input data of the UDF, however, this will not work in all scenarios because the queries themselves are not required to be constant and might depend on external data. Instead, the actual loopback queries must be issued to the database when the UDF is running. This can be done by maintaining an open client connection to the database during client-local execution.

Extending to Other Databases

Our solution is implemented for MonetDB/Python. The same implementation strategy can be used for other RDBMSes by extending their respective clients. However, the processing model of the respective database needs to be taken into consideration. MonetDB uses the operator-at-a-time processing model, which means the UDFs are only called once with the entire columns as input. Row-store databases (e.g. Postgres or MySQL) use the tuple-at-a-time processing model, under which the UDFs are called many times with only individual rows as input. As this changes the way UDFs are called, the debugging framework must be adapted to these differing processing models.

4. Conclusions & Future Work

In this paper, we identify the problem of debugging scripting language UDFs in databases and propose two different solutions to allow developers to quickly and easily debug UDFs using modern techniques. The developer provides a problematic SQL query in which the UDF is executed, after which we export the UDF and any required data from the database so it can run locally in an interactive debugger on the developers' machine.

Future Work.

In our current solution, the UDFs and required data are exported so they can be debugged outside the database. However, this introduces both security and performance issues. A way to solve this issue is to run the debugger directly in the database. Another possibility of future work would be the development of an IDE plugin that easily allows the user to export, import and modify his UDFs.

Acknowledgments

This work was funded by the Netherlands Organisation for Scientific Research (NWO), projects "Data mining on high volume simulation output" (Holanda) and "Process mining for multi-objective online control" (Raasveldt).

References

Hailpern, B. and Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12.

McConnell, S. (2004). *Code complete*. Pearson Education.

Pérez, F. and Granger, B. E. (2007). IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29.

Raasveldt, M. and Mühleisen, H. (2016). Vectorized udfs in column-stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20, 2016*, pages 16:1–16:12.

Raasveldt, M. and Mühleisen, H. (2017). Dont hold my data hostage-a case for client protocol redesign. *Proceedings of the VLDB Endowment*, 10(10):1022–1033.