Integrating analytics with relational databases

Raasveldt, M.

CHAPTER 4

Vectorized UDFs in Column-Stores

1 Introduction

In Chapter 2, we described how in-database processing can be used to mitigate the overhead of exporting data from the database server. In this chapter, we dive further into using in-database processing for analytics by looking at user-defined functions. Specifically, we focus on user-defined functions in interpreted languages such as R, Python or MATLAB, which are the most commonly used languages in data science [47].

These languages, which we call vector-based languages, provide additional challenges when used in user-defined functions. If we were to simply use them as a one–to–one replacement for compiled languages such as C or Java the functions will have very poor performance. While compiled languages are very efficient when operating on individual elements, these interpreted languages are not. In interpreted languages actions that are normally performed while compiling, such as type checking, are performed at run-time. This interpreter overhead is performed before every operation, even before simple operations such as addition or multiplication. For many of these operations, this overhead dominates the actual cost of the operation. As a

result, operations performed on individual elements are very inefficient.

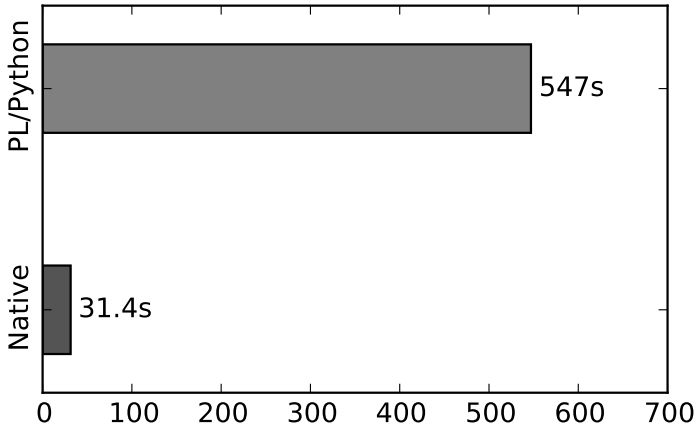


Figure 4-1: Modulo computation in Postgres.

This issue is demonstrated in Figure 4-1, where we compute the modulo of 1 GB of integers using both Postgres' built-in modulo function and a Python UDF in Postgres. We can see that the interpreter overhead results in the Python UDF taking much longer to perform the exact same operation.

These interpreted languages rely on *vectorized operations* for efficiency. Rather than operating on individual values, these operations process arrays directly. When using these vectorized operations the interpreter overhead is only incurred once for every array, rather than once for every value. By using vectorized operations they can process data as efficiently as compiled languages. However, we can only use these vectorized operations if we have access to chunks of the data at the same time. This does not fit into the way user-defined functions are typically processed in databases. Rather than processing one row at a time, they have to process multiple rows or even entire tables at the same time to operate efficiently.

1.1 Contributions

In this chapter we discuss how vector-based languages can be integrated into various database processing engines, and how various database architectures influence the performance of user-defined functions in vector-based languages. We describe our

system, MonetDB/Python, that efficiently integrates vectorized user-defined functions into the open-source database MonetDB. We describe how these user-defined functions fit into the processing model of the database, and show how these functions can be automatically parallelized by the query execution engine of the database server. We compare the performance of our implementation with in-database processing solutions of alternative open-source database systems, and demonstrate the efficiency of vectorized user-defined functions. We show that vectorized user-defined functions in interpreted languages can be as fast as user-defined functions written in compiled languages, without requiring any in-depth knowledge of database kernels and without needing to compile and link them to the database server. MonetDB/Python is open-source. The source code is freely available online in the official MonetDB source code repository [1].

1.2 Outline

This chapter is organized as follows. In Section 2, we review different types of user-defined functions. In Section 3, we present MonetDB/Python. In Section 4, we show the results of a set of benchmarks that compare the performance MonetDB/Python functions against user-defined functions in different languages and different databases. In Section 5, we present related work. In Section 6, we describe how our work could be applied to other databases. We describe our efforts into improving the development workflow of MonetDB/Python UDFs in Section 7. Finally, in Section 8, we draw our conclusions.

2 Types of User-Defined Functions

Before we discuss the implementation of our user-defined functions, we will first briefly discuss the different types of user-defined functions in this section.

User-Defined Scalar Functions are *n-to-n* operations that operate on an arbitrary number of input columns and output a single column. These functions can be

[1] https://dev.monetdb.org/hg

used in the *SELECT* and *WHERE* clauses of a SQL query. An example of a simple scalar user-defined function is one that imitates the functionality of the multiplication operator: it takes as input two columns, and outputs a single column that results from multiplying the input columns together.

User-Defined Aggregate Functions are *n-to-g* operations that perform some aggregation on the input columns, possibly over a number of groups with the `GROUP BY` statement. These can be used in the *SELECT* and *HAVING* clauses of a SQL query. An example of a user-defined aggregate function is a function that emulates the `MAX` function, that returns the maximum of all the values in a column.

User-Defined Table Functions are operations that do not return a single column, but rather return an entire table with an arbitrary number of columns. These can be used in the *FROM* clause of a SQL query. The possible input of table producing functions vary depending on the database. Certain databases only support the input of scalar values, whereas others support the input of other tables. In MonetDB, the input of a user-defined table function can come from a subquery, and hence the input of a user-defined table function can be any table.

3 MonetDB/Python

In this section we describe the internal pipeline of MonetDB/Python functions. We describe how the data is converted from the internal database format to a format usable in Python, and how these functions are parallelized.

3.1 Usage

As MonetDB/Python functions are interpreted, they do not need to be compiled or linked against the database. They can be created from the SQL interface and can be immediately used after being created. The syntax for creating a MonetDB/Python function is shown in Listing 4.1.

```
CREATE FUNCTION fname([paramlist | *])
RETURNS [TABLE(paramlist) | returntype]
LANGUAGE [PYTHON | PYTHON_MAP]
[{ functioncode } | 'external_file.py'];
```

Listing 4.1: MonetDB/Python Syntax.

A MonetDB/Python function can be either a user-defined scalar, aggregate or a table function. A user-defined scalar function takes an arbitrary number of columns as input and returns a single column, and can be used anywhere a normal SQL function can be used. A user-defined aggregate function also outputs a single column, but can be used to process aggregates over several groups when a `GROUP BY` statement is present in the query. A user-defined table function can take an arbitrary number of columns as input and can return an entire table. User-defined table functions can be used anywhere a table can be used.

```
CREATE FUNCTION pysqrt(i INTEGER)
RETURNS REAL
LANGUAGE PYTHON {
    return numpy.sqrt(i)
};

SELECT pysqrt(i * 2) FROM tbl;
```

Listing 4.2: Simple Scalar UDF.

An example of a scalar function that computes the square root of a set of integers is given in Listing 4.2. Note that the function is only called once, and that the variable `i` is an array that contains all the integers of the input column. The output of the function is an array containing the square root of each of the input values.

3.2 Processing Pipeline

MonetDB/Python functions are executed as an operator in the processing model of the database, as illustrated in Figure 4-2. MonetDB/Python functions run in the same process and memory space as the database server. As such, MonetDB/Python

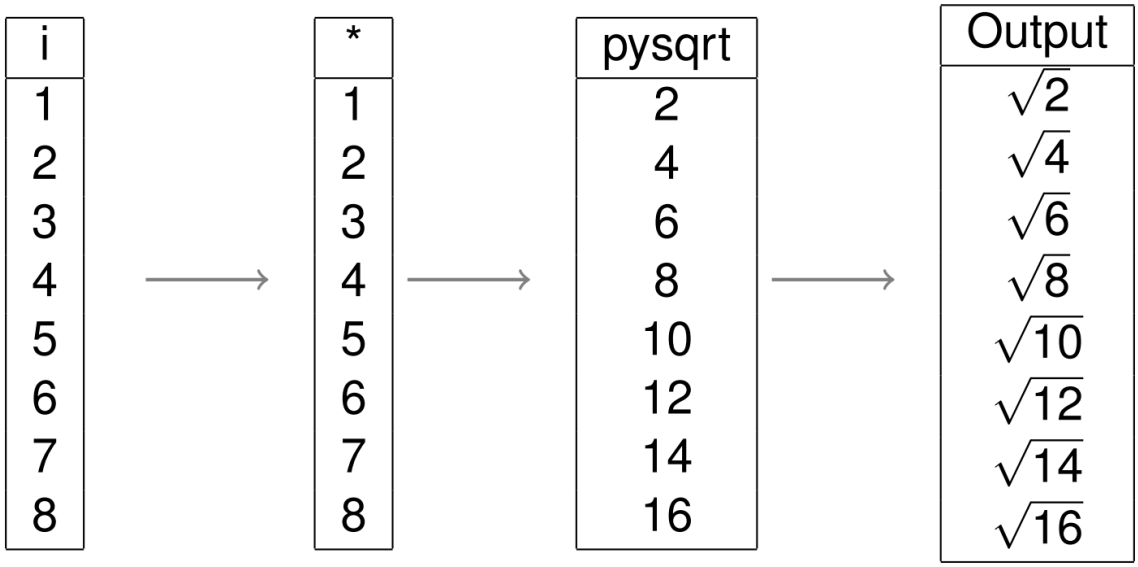


Figure 4-2: Operator Chain for Listing 4.2.

functions behave identically to other operators in the operator–at–a–time processing model. MonetDB/Python functions are called once with a set of columns as input, and must return a set of columns as output.

The general pipeline of the MonetDB/Python functions is as follows: first, we have to convert the input columns to a set of Python objects. Then, we execute the stored Python function with the converted columns as input. Finally, we convert the resulting Python objects back to a set of database columns which we then hand back to the database.

Input Conversion. The database and the interpreted language represent data in a different way. As such, the data has to be converted from the format used by the database to a format that works in the interpreted language. Data conversion can be an expensive operation, especially when a large amount of data has to be converted. Unfortunately, we cannot avoid data conversion when writing a user-defined function in a different language than the core database language.

Since MonetDB is a main-memory database, the database server keeps hot columns loaded in main memory. As MonetDB/Python functions run in the same memory space as the database server we can directly access the columns that are loaded in memory. As a result, the only cost we have to pay to access the data is the cost for

converting this data from the databases' representation to a representation usable in Python.

Internally, columns in a column-store database are very similar to arrays. They hold a list of elements of a single type, one element for every row in the table. As such, the most efficient uncompressed representation for a column is a tightly packed array where the elements are stored subsequently in memory. By using this representation, each element of n bytes occupies exactly n bytes.

MonetDB represents the data of individual columns as tightly packed arrays. In addition to the actual data, the columns contain metadata, such as the type of the column and whether or not the column contains null values.

Vector-based languages work with arrays containing a single type as well. As such, they have the exact same optimal data representation as columns in a column-store database. It should then be no surprise that the data in both NumPy arrays and R vectors are also internally represented as tightly packed arrays.

As both the database and the vector-based language share the same representation for the data, we do not need to convert the data values. Instead, all we have to convert is a small amount of metadata before we can use the databases' columns in Python. As we are not touching the actual data, the input conversion costs a constant amount of time.

Code Execution. After converting the input columns to a set of Python objects, the actual user-defined function is interpreted and executed with the set of Python objects as input. The user can then use Python to manipulate the input objects and return a set of output objects.

Aside from the parallel processing, which is described in Section 3.3, we do not perform any optimization on the users' code. That means that the interpreter overhead depends entirely on the code created by the user. If the user calls a constant amount of vectorized functions, the interpreter overhead is constant. As vector-based languages are only efficient when vectorized functions are used, this is expected to be a common scenario.

On the other hand, if the user calls functions that operate on the individual

elements of the data, the interpreter overhead scales with the amount of function calls and can become a serious bottleneck.

Output Conversion. The database expects a set of columns as output from the user-defined function. As such, the same conversion method can be used to convert vectors back to database columns, but in reverse. Instead of directly using the data from the database, we take the data from the returned set of vectors and convert it to a set of columns in the database. Again, we only need to convert the necessary metadata, leading to a constant conversion time.

Total Overhead. As MonetDB/Python functions are not written in the databases' native language, they incur overhead for converting between different object representations. In addition, as Python is an interpreted language, the functions incur additional interpreter overhead as well.

The conversion overhead only costs a constant amount of time for each function call as we only convert the metadata, and this overhead is only incurred once for each time the function is called in a SQL statement. This overhead would be significant for transactional workloads, where the function could be called many times with only a small amount of data as input. However, as both MonetDB and NumPy are designed around analytical workloads, we do not expect transactional workloads. For analytical workloads that operate on large chunks of data, this constant amount of overhead is not significant.

The magnitude of the interpreter overhead depends entirely code written by the user. If scalar functions are used, the interpreter overhead can dominate the computation time. However, when the code only calls a constant amount of vectorized functions, the interpreter overhead is constant as well. In this case, the performance of MonetDB/Python UDFs is comparable to a UDF written in the databases' native language, as illustrated in Figure 4-5.

3.3 Parallel Processing

In Section 3.2 we discussed the efficient conversion of data from the format used by the database to the format used by Python. The efficient data transfer from the

database to Python significantly improves the performance of functions for which the data transfer and conversion is the main bottleneck. However, the Python function is still executed by the regular Python interpreter. As such, the efficient data conversion does not significantly improve the performance of functions that are bound by the Python execution time.

Users can manually improve the performance of these functions by executing them in parallel. However, we would prefer to not push the burden of optimization onto the user. In addition, manual parallelization of user-defined functions can result in conflicts with the workload management of the database, which can significantly decrease database throughput [90]. It would be preferable to have the parallelization handled automatically by the database server. However, there are several issues with automatic parallelization in the database processing pipeline.

```
1 SELECT MEDIAN(SQRT(i * 2)) FROM tbl;
```

Listing 4.3: Chain of SQL operators.

In an operator–at–a–time database, the operators are only called once. How do we move to a model where data is processed in parallel? The solution employed by MonetDB is to split up the columns into separate chunks and call the parallelizable operators once for every chunk. The non-parallelizable operators, such as the median, force the chunks to be packed together into a single array and are then called with that entire array as input. This process is shown in Figure 4-3.

While the figure displays a table with eight entries split up into four parts as an example, small columns are normally not split into separate chunks as the additional multithreading overhead would be larger than the time saved by parallelizing the query. Instead, a heuristic is used to determine when columns should be split up based on the size of the columns.

MonetDB/Python functions can be automatically parallelized in this system as well. This alleviates the burden of parallelization from the user, and leaves the database in full control of the parallelization. However, not all functions can be automatically parallelized in this format. A user-defined function that computes the median, for example, requires access to all the data in the column.

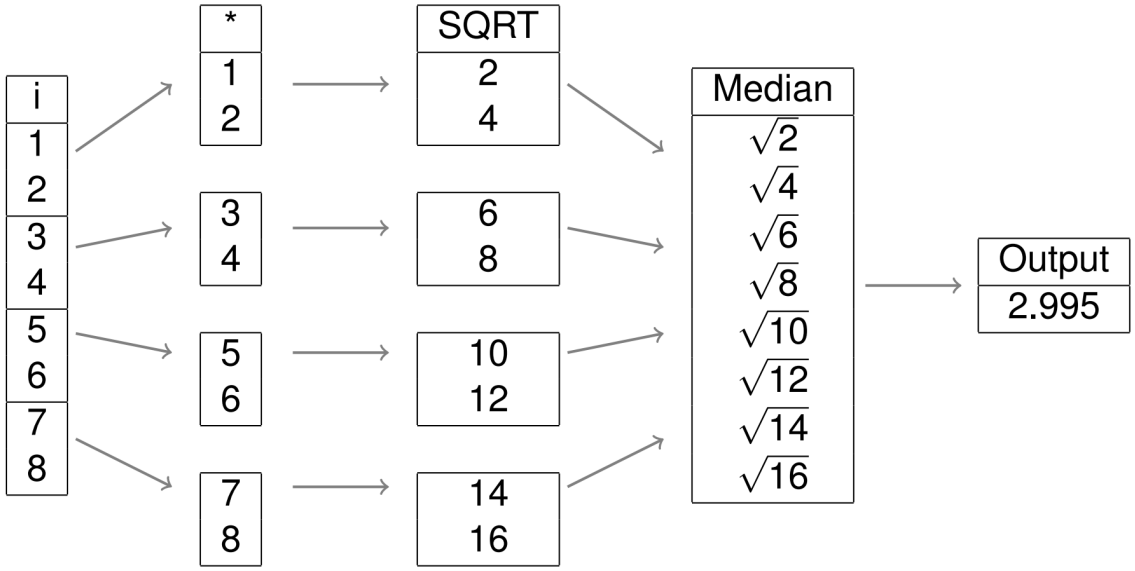


Figure 4-3: Parallel Operator Chain of Listing 4.3.

As such, we require the user to specify whether or not their UDF can be executed in parallel when creating the function. When the function cannot be run in parallel, it will run as a blocking operator and get access to the entire input columns. This behavior is identical to the median computation seen in Figure 4-3.

Parallel computation has an additional effect on the function call overhead of MonetDB/Python functions as we are no longer only calling parallel functions once. The functions are called once per chunk, meaning the function call overhead is incurred once per chunk.

The amount of chunks created is at most equal to the amount of virtual cores that the system has, meaning the function call overhead is $O(p)$ instead of $O(1)$, where p is the amount of cores. However, as the input columns are only split up when they have a sufficient size, this additional overhead will never dominate the actual computation time.

Chaining Operators. Operating on partitions of the data is a straightforward way of parallelizing operators. However, as these partitions are arbitrary, the operators can only be parallelized if they are completely independent and only operate on individual rows. As such. many operators cannot be completely parallelized in this

fashion.

Often, operators can only be partially computed in parallel, and require a final step that merges the results of the parallel computation to create the final result. An example of such an operator is the `sort` operator. The chunks can be sorted in parallel, but will then have to be merged together to fully sort the column.

```
1 SELECT minseq(minmap(i)) FROM tbl;
```

Listing 4.4: Parallel MIN using chained operators.

We can parallelize these operators in our system by chaining together operators in the SQL layer. The parallel component of the operator can be computed in a mappable function. The output columns of the parallel components can then be passed to a blocking function, which merges these columns together to create the final result. An example of such a chain being used to compute the minimum value of a column in parallel is given in Figure 4-4.

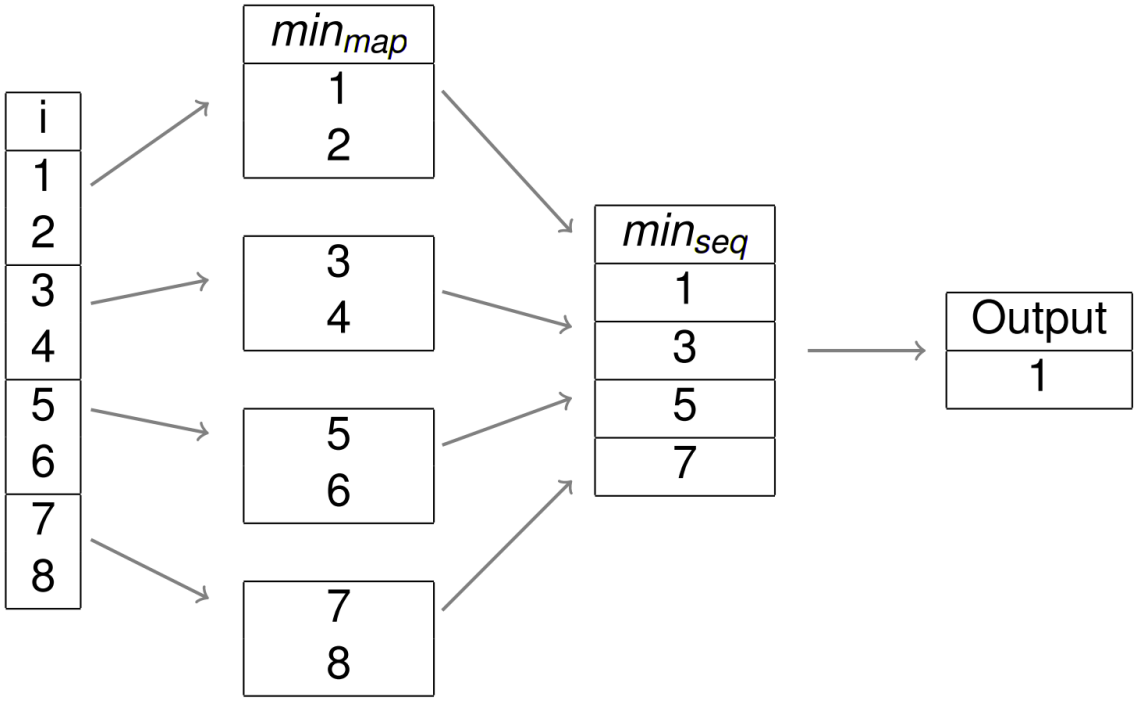


Figure 4-4: Operator Chain of Listing 4.4.

User-defined table functions can be chained together in a similar but more flexible way. These operators can take entire tables as input and output entire tables of

arbitrary size. Chaining these operators together allows many different operations to be executed in parallel.

Parallel Aggregates. The parallel processing we have implemented operates on sequential segments of the data. If a column is partitioned into two parts, the first partition will hold the first half of all the values in the column, and the second part will hold the second half. The reason we use this partitioning scheme is the virtual identifiers used by MonetDB. Any other partitioning requires us to explicitly keep track of the individual identifiers. By using sequential partitioning we do not need to materialize the identifiers of the rows, as the statement that entry i in the column corresponds to row $oid_{base} + i$ still holds.

Parallel computation of aggregates is a special case where we can split up the data into arbitrary partitions without needing to materialize the row identifiers. This is because when we compute the aggregates over several groups, the only information we need is to which group a specific entry belongs. We do not need to know to which specific row it belongs. As such, rather than using sequential partitions we can create one separate partition for each group. We can then compute the separate aggregates for each group in parallel by calling the UDF once per group partition.

The problem with this scheme is that the interpreter overhead is incurred once per group, and the amount of groups can potentially be very large. In the most extreme case, the amount of groups is equal to the amount of tuples in the input columns. In this case, we incur the interpreter overhead once for every tuple.

We can avoid this potentially large interpreter overhead by allowing the user to compute more than one aggregation per function call. To do this, the function has to know the group that each tuple belongs to in the aggregation. We can pass this to the user-defined function as an additional input column. The user can then perform the aggregation over each of the different groups, and return the aggregated results in order.

These functions can be parallelized in a similar manner. We can split the data into different sets, where each set contains all the data of a number of groups and the corresponding group identifiers of each tuple.

3.4 Loopback Queries

MonetDB/Python also 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 Python objects in a similar way as the input of the UDFs is converted. 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 4.5 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.

```
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) }
};
```

Listing 4.5: Loopback Queries

4 Evaluation

In this section we describe a set of experiments that we have run to test how efficient MonetDB/Python is compared to alternative in-database processing solutions.

The experiments were run on a machine with two Intel Xeon (E5-2650 v2) 2.6GHZ CPUs, with a total of 16 physical and 32 virtual cores and 256 GB RAM. The machine uses the Fedora 20 OS, with Python version 2.7.5 and NumPy version v1.10.4. The measured time is the wall-clock time for the completion of the query.

For each of the benchmarks, we ran the query five times, which was sufficient for the standard deviation to converge. The result displayed in the graph is the mean

of these measured values. All benchmarks performed are hot tests. We first ran the query twice to warm up the database prior to running the measured runs.

4.1 Systems Measured

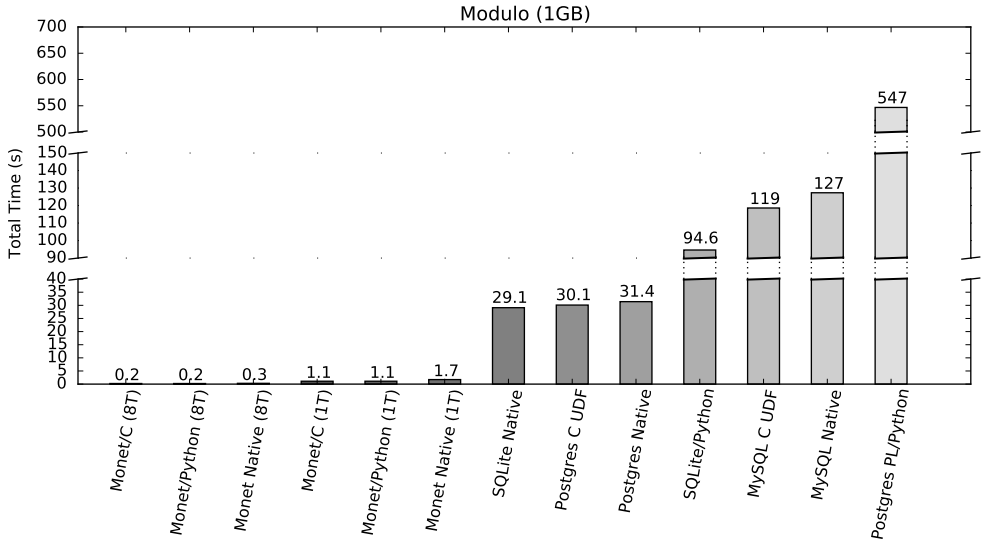


Figure 4-5: Modulo computation of 1GB of integers.

MySQL is the most popular open-source relational data-base system. It is a row-store database that is optimized for OLTP queries, rather than for analytical queries. MySQL supports user-defined functions in the languages *C/C++* [79].

Postgres is the second most popular open-source relational database system. It is a row store database that focuses on being SQL compliant and having a large feature set. Postgres supports user-defined functions in a wide variety of languages, including *C*, Python, Java, PHP, Perl, R and Ruby [67].

SQLite is the most popular embedded database. It is a row-store database that can run embedded in a large variety of languages, and is included in Python's base library as the *sqlite3* package. SQLite supports user-defined functions in *C* [23], however, there are wrappers that allow users to create scalar Python UDFs as well.

MonetDB is the most popular open-source column-store relational database. It is focused on fast analytical queries. MonetDB supports user-defined functions in the languages *C* and *R*, in addition to MonetDB/Python.

We want to investigate how efficient the user-defined functions of these different databases are, and how they compare against the performance of built-in functions of the database. In addition, we want to find out how efficient MonetDB/Python is compared to these alternatives.

4.2 Modulo Benchmark

In this benchmark, we are mainly interested in how efficiently the data is transported to and from the user-defined functions. As we have seen in Figure 4-1, this is a crucial bottleneck for user-defined functions.

We will compute the modulo of a set of integers in each of the databases. The modulo is a good fit for this benchmark for several reasons: unlike floating point operations such as the `sqrt`, there is no estimation involved. When estimation is involved, the comparison is often not fair because a system can estimate to certain degrees of precision. Naturally, more accurate estimations are more expensive. However, in a benchmark we would only measure the amount of time elapsed, thus the more accurate estimation would be unfairly penalized.

Similarly, when performing a modulo operation, we know that there is a specific bound on the result. The result of $x \% n$ will never be bigger than n. This means that there is no need to promote integral values. If we were to compute multiplication, for example, the database could be promoting `INT` types to `LONGINT` types to reduce the risk of integer overflows. This naturally takes more time, and could make benchmark comparisons involving multiplication unfair.

In addition, the modulo operation is a simple scalar operation that can be easily implemented in both *C* and NumPy by using the modulo operator. This means that we will not be benchmarking different implementations of the same function, but we will be benchmarking the efficiency of the database and data flow around the function. As it is a simple scalar operation, it also fits naturally into *tuple-at-a-time* databases. We can also trivially compute the modulo operation in parallel, allowing us to benchmark the efficiency of our parallel execution model.

Setup. In this benchmark, we computed modulo `100` of `1GB` of randomly generated

32-bit integers. The values of the integers are uniformly generated between the values 0 and 2^{31}. To ensure a fair comparison, every run uses the same set of values. For each of the mentioned databases, we have implemented user-defined functions in a subset of the supported UDF languages to compute the modulo. In addition, we have computed the modulo using the built-in modulo function of each database. For MonetDB, we have measured both the multi-threaded computation (with 8 threads) and the single-threaded computation.

Results. The results of the benchmark are shown in Figure 4-5. As we can see, MonetDB provides the fastest computation of the modulo. This is surprising, considering the modulo function is well suited for *tuple-at-a-time* processing. In addition, the table we used had no unused columns. It only had a single column containing the set of integers, thus this is essentially a best-case scenario for the tuple-at-a-time databases.

The reason for this performance deficit is that even when computing scalar functions, the function call overhead for every individual row in the data set is very expensive when working with a large amount of rows. When the data fits in memory, the *operator-at-a-time* processing of MonetDB provides superior performance, even though access to the entire column is not necessary for the actual operators.

We note that in all of the databases our user-defined functions in *C* are faster than the built-in modulo operator. This is because our user-defined functions skip sanity checks that the built-in operators perform, such as checking for potential *null* values that could be in the database, and instead directly compute the modulo. This allows our user-defined functions to be faster than the built-in operators on all database systems.

When looking at the Python UDFs, we immediately note the additional interpreter overhead that is incurred in the tuple-at-a-time databases. Both SQLite/Python and PL/Python have poor performance compared to the native modulo operator in their respective database. In these architectures, the user-defined functions are called once per row, which incurs a severe performance penalty. We note that PL/Python is significantly slower than SQLite/Python. This is because SQLite/Python is a very

thin wrapper around C UDFs that minimize overhead, while PL/Python offers more complex functionality which cause these functions to incur significantly more overhead.

By contrast, MonetDB/Python is just as fast as the UDF written in *C* in MonetDB. Because of our vectorized approach, the conversion and interpreter overhead that MonetDB/Python UDFs incur is minimal. As such, they achieve the same performance as UDFs written in the databases' native language, but without requiring the user to have in-depth knowledge of the database kernel and without needing to compile and link the function to the database.

5 Related Work

There is a large body of related work on user-defined functions, both in the research field and in implementations by database vendors. In this section, we will present the relevant related work in both fields, and compare the related work against MonetDB/Python.

5.1 Research

Research on user-defined functions started long before they were introduced into the SQL standard. The work by Linnemann et al. [52] focuses on the necessity of user-defined functions and user-defined types in databases, noting that the SQL standard lacks many necessary functions such as the square root function. To solve this issue, they suggest adding user-defined functions, so the user can add any required functions themselves. They describe their own implementation of user-defined functions in the compiled PASCAL language, noting that the compiled language is nearly as efficient as built-in functions, with the only overhead being the conversion costs.

They note that executing UDFs in a low-level compiled language in the same address space as the database server is potentially dangerous. Mistakes made by the user in the UDF can corrupt the data or crash the database server. They propose two separate solutions for this issue; the first is executing the user-defined function in a separate address space. This prevents the user-defined function from accessing the

memory of the database server, although this will increase transfer costs of the data.

The second solution is allowing users to create user-defined functions in an interpreted language, rather than a low-level compiled language, as interpreted languages do not have direct access to the memory of the database server. This is exactly what MonetDB/Python UDFs accomplish. By running in a scripting language, they can safely run in the same address space as the database and avoid unnecessary transfer overhead.

In-Database Analytics

In-database processing and analytics have seen a big surge in popularity recently, as data analytics has become more and more crucial to many businesses. As such, a significant body of recent work focuses on efficient in-database analytics using user-defined functions.

The work by Chen et al. [14, 15] takes an in-depth look at user-defined functions in tuple-at-a-time processing databases. They note that while user-defined functions are a very useful tool for performing in-database analysis without transferring data to an external application, existing implementations have several limitations that make them difficult to use for data analysis. They note that existing user-defined functions in *C* are either very inefficient compared to built-in functions, as in SQL Server, or require extensive knowledge of the internal data structures and memory management of the database to create, as in Postgres, which prevents most users from using them effectively. MonetDB/Python UDFs do not have this issue, as they do not require the user to have in-depth knowledge of the database internals.

They also identify issues with user-defined functions in popular databases that restrict their usage for modeling complex algorithms. While user-defined scalar functions and user-defined aggregate functions cannot return a set, user-defined table functions cannot take a table as input in the database systems they used. The same observation is made by Jaedicke et al. [45]. The result of this is that it is not possible to chain multiple user-defined functions together to model complex operations, that each take a relation as input and output another relation.

To alleviate this issue, both Chen et al. [14] and Jaedicke et al. [45] propose a new set of user-defined functions that can take a relation as input and produce a relation as output. This is exactly what MonetDB/Python table functions are capable of. They can take an arbitrary number of columns as input and produce an arbitrary number of columns as output, and can be chained together to model complex relations.

The work by Sundlöf [78] explores the difference between performing computations in-database with user-defined functions and performing the computations in a separate application, transferring the data to the application using an ODBC connection. Various benchmarks were performed, including matrix multiplication, singular value decomposition and incremental matrix factorization. They were performed in the column-store database Sybase IQ in the language *C++*. The results of his experiments showed that user-defined functions were up to thirty times as fast for computations in which data transfer was the main bottleneck.

Sundlöf noted that one of the difficulties in performing matrix operations using user-defined functions was that all the input columns must be specified at compile time. As a result it was not possible to make user-defined functions for generic matrix operations, but instead they had to either create a separate set of user-defined functions for every possible amount of columns, or change the way matrices are stored in the database to a triplet format *(row number, column number, value)*.

Processing of User-Defined Functions

As user-defined functions form such a central role in in-database processing, finding ways to process them more efficiently is an important objective. However, as the user-defined functions are entirely implemented by the user, it is difficult to optimize them. Nevertheless, there has been a significant effort to optimize the processing of user-defined functions.

Parallel Execution of User-Defined Functions

Databases can hold very large data sets, and a key element in efficiently processing these data sets is processing them in parallel, either on multiple cores or on a cluster

of multiple machines. Since user-defined functions can be very expensive, processing them in parallel can significantly boost the performance of in-database analytics. However, as user-defined functions are written by the user themselves, automatically processing them in parallel is challenging.

The work by Jaedicke et al. [44] explores how user-defined aggregate functions can be processed in parallel. They require the user to specify two separate functions, a local aggregation function and a global aggregation function. The local aggregation function is executed in parallel on different partitions of the data. The results of the local aggregation functions are then gathered and passed to the global aggregation function, which returns the actual aggregation result.

They propose a system that allows the user to define how the data is partitioned and spread to the local aggregation functions. More strict partitions are more expensive to create, but allow for a wider variety of operations to be executed in parallel.

5.2 Systems

In this section, we will present an overview of systems that have implemented user-defined functions. We will take an in-depth look at the types of user-defined functions these systems support, and how they differ from MonetDB/Python.

Aster nCluster Database

The Aster nCluster Database is a commercial database optimized for data warehousing and analytics over a large number of machines. It offers support for in-database processing through *SQL/MapReduce functions* [30]. These functions support a wide set of languages, including compiled languages (C++, C and Java) and scripting languages (R, Python and Ruby).

SQL/MR functions are parallelizable. As in the work by Jaedicke et al. [44], they allow users to define a partition over the data. They then run the SQL/MapReduce functions in parallel over the specified set of partitions, either over a cluster of machines or over a set of CPU cores.

SQL/MR functions support polymorphism. Instead of specifying the input and output types when the function is created, the user must provide a constructor for the user-defined function. The constructor takes as input a *contract* that contains the input columns of the function. The constructor must then check if these input columns are valid, and provide a set of output columns. During query planning, this constructor is called to determine the input/output columns of the SQL/MR function, and a potential error is thrown if the input/output columns do not line up correctly in the query flow.

The primary difference between SQL/MR functions and MonetDB/Python functions is the processing model around which they are designed. SQL/MR functions operate on individual tuples in a *tuple-at-a-time* fashion. The user obtains the next row by calling the `advanceToNextRow` function, and outputs a row using the `emitRow` function.

6 Applicability To Other Systems

In the paper, we have described how we integrated user-defined functions in a vector-based language in the operator-at-a-time processing model. In this section, we will discuss how functions in vector-based languages could be efficiently integrated into different processing models.

Tuple–at–a–Time. We have already determined that the straightforward implementation of vector-based language UDFs in this processing model is very inefficient. When a vector-based language is used to compute scalar values, the interpreter overhead dominates the actual computation cost. Instead, the UDF should receive a large chunk of the input to operate on so the interpreter overhead is negligible compared to the actual computation cost.

In the tuple-at-a-time processing model, accessing a chunk of the input at the same time requires us to iterate over the tuples one by one. Then, after every value has been computed, we copy that value to a separate location in memory. After gathering a set of values, we can use the accumulated array of values as input values for a vectorized

UDF.

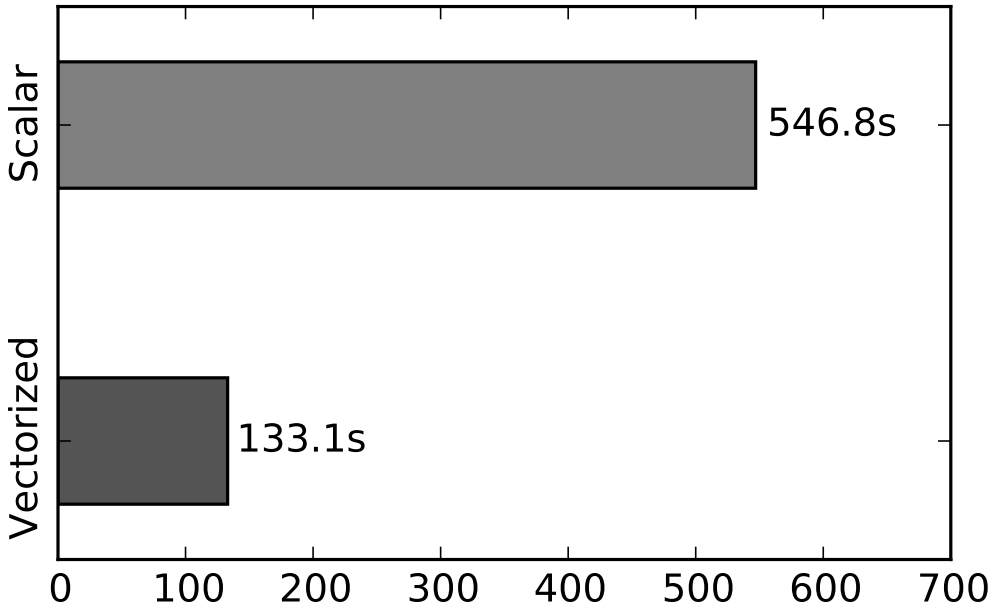


Figure 4-6: PL/Python Vectorized vs Non-Vectorized Modulo Operator.

While gathering the data requires additional work, this added overhead is significantly lower than the interpreter overhead incurred when operating on scalar values in a vector-based language. This is especially true when a lot of different operations are performed on the data in the UDF.

We have emulated this algorithm in Postgres by loading the data of a single column into PL/Python using a database access function, and then calling the vector based operator on the entire column at once. The results are shown in Figure 4-6. We can see that this method is significantly more efficient than performing many scalar operations even when we perform only a single operation (modulo).

However, this method is still significantly slower than MonetDB/Python because of the added overhead for copying and moving the data. As such, it is not possible for vector-based languages to perform as efficiently as native database functions in this processing model.

Vectorized Processing is similar to our parallel processing model. It operates on chunks of the data. Parallel UDFs fit directly into this processing model in a

similar fashion. They would operate on one chunk at a time, and incur the interpreter overhead once per chunk. The magnitude of the interpreter overhead depends entirely on the size of the chunks. While MonetDB/Python always operates on chunks with a high cardinality, this is not necessarily true in databases with vectorized processing. If the chunks sizes are too small, then the interpreter overhead will still dominate the processing time.

Blocking UDFs in this processing model have the same issues as UDFs in the tuple-at-a-time processing model. The UDF needs access to all the input data at once, but the database only computes the data in chunks. As such, we need to gather the data from each of the separate chunks before calling the blocking function. In the operator-at-a-time processing model, this is only necessary if the blocking function is executed after a paralellized function.

Compressed Data. Certain databases work with compressed data internally to save storage space and memory bandwidth. Especially column-oriented database systems can benefit greatly from compression. When the input columns to a vector-based UDF are compressed, they have to be entirely decompressed before being passed to the vector-based function, unless the vector-based language itself supports the processing of compressed columns.

7 Development Workflow: devUDF

The generic workflow for developing a UDF is to write a function using a simplistic text editor. The function can then be created inside the RDBMS through a SQL command, and used by calling it within a SQL query. If there are bugs or problems within the UDF, the function has to be recreated and the SQL query has to be rerun. This process has to be repeated until the problem is fixed.

This workflow is problematic when developing complex UDFs, as advanced IDE features and modern debugging techniques cannot be used. Using these IDE features is not easily doable because the developer has to manually perform code transformations to convert the Python code to a SQL command that creates the UDF. As seen in

Table 4.1 [11], IDEs are heavily preferred for development over simplistic text editors due to their development features. Therefore, we argue that offering support for the usage of these features in the development workflow of UDFs will make developing UDFs more attractive, faster and easier for many developers.

Name	Market Share	Type
`Eclipse`	25.2%	**IDE**
`Visual Studio`	19.5%	**IDE**
`Android Studio`	9.5%	**IDE**
`Vim`	7.9%	Text Editor
`XCode`	5.2%	**IDE**
`IntelliJ`	4.8%	**IDE**
`NetBeans`	4.0%	**IDE**
`Xamarin`	3.8%	**IDE**
`Komodo`	3.4%	**IDE**
`Sublime Text`	3.3%	Text Editor
`Visual Studio Code`	3.3%	Text Editor
`PyCharm`	2.3%	**IDE**

Table 4.1: Most Popular Development Environments.

IDEs are also attractive because they facilitate the usage of sophisticated interactive debugging techniques, such as stepping through the code line by line and pausing code execution. However, these techniques cannot be used in conjunction with UDFs because the RDBMS must be in control of the code flow while the UDF is being executed. Instead, developers have to resort to inefficient debugging strategies (e.g., print debugging) to make their code work [40].

Another issue with the standard UDF workflow is that UDFs are stored within the database server. As a result, version control systems (VCSs) such as Git [53] cannot be easily integrated to keep track of changes to UDFs. Without a VCS, cooperative development is challenging and the development history is not stored.

For the purpose of enhancing development efficiency for UDFs, we developed devUDF, a plugin for the popular IDE PyCharm that facilitates developing and debugging MonetDB/Python UDFs directly from within the IDE. Using our plugin, advanced debugging features can be used while refining and refactoring UDFs.

7.1 The devUDF Plugin

The devUDF plugin is developed for the PyCharm IDE that facilitates the usage of advanced IDE features for development of MonetDB/Python UDFs. It allows developers to create, modify and test UDFs without leaving their IDE environment. All features of the IDE can be used to develop UDFs, including the sophisticated interactive debugger and VCS support.

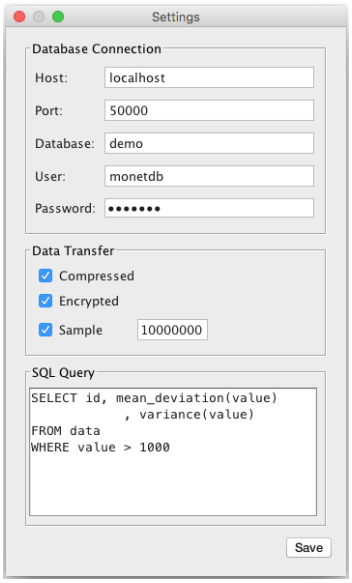


Figure 4-7: Settings.

7.2 Usage

The devUDF plugin can be accessed through the main menu of the IDE (See Figure 4-8). In this menu, a submenu labeled "UDF Development" contains the three main aspects of the plugin.

Initially, devUDF must be configured so it can connect to an existing database

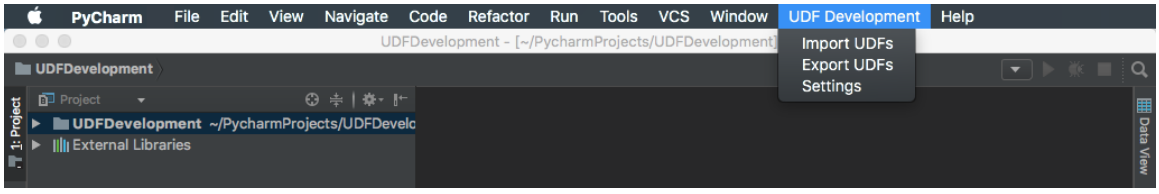
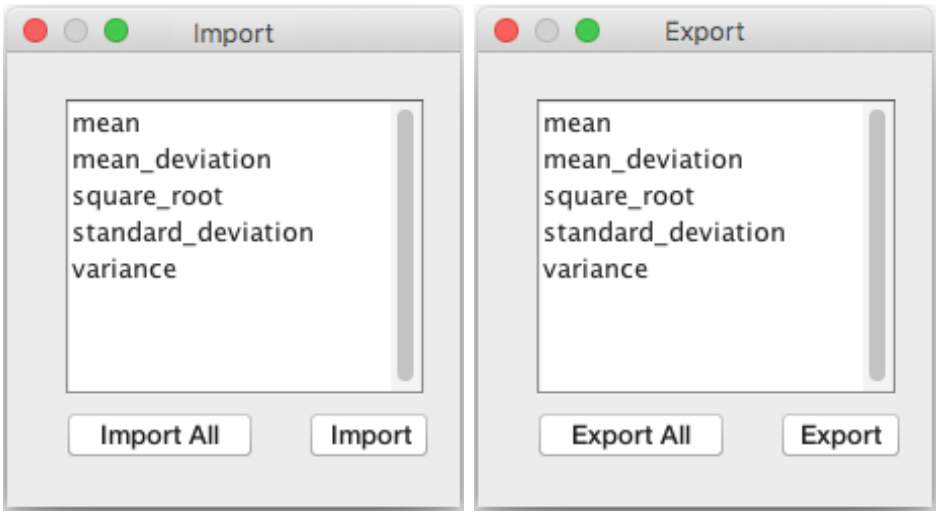


Figure 4-8: PyCharm Main Menu.



(a) Import (b) Export.

Figure 4-9: Importing and Exporting UDFs from the Database.

server. This can be done through the settings window shown in Figure 4-7. The parameters required are the usual database client connection parameters (i.e., host, port, database, user and password).

After the devUDF plugin has been configured to connect to a running database server, the development process begins by importing the existing UDFs within the server into the development environment. This is done through the "Import UDFs" window, shown in Figure 4-9a. The developer has the option to select the functions that he wishes to import, or he can choose to import all functions stored within the database server.

After the UDFs are imported, the code of the UDFs is exported from the database and imported into the IDE as a set of files in the current project. The developer can then modify the code of the UDFs in these files, use version control to keep track of changes to the UDFs and export the UDFs back to the database server for execution through the "Export UDFs" window (see Figure 4-9b).

The developer can also run any of the imported UDFs with the IDEs interactive debugger by running the project as they would run a normal PyCharm project (using the "Debug" command). Since a UDF is never executed in isolation, but always within the context of a SQL query, the user must provide a SQL query which executes the

to-be-debugged UDF. This SQL query must be specified in the Settings menu (see Figure 4-7).

Running the UDF in the interactive debugger will execute the function locally on the developers' machine instead of remotely inside the database server. As the UDF requires data from the database (as its input parameters), the data must be transferred from the database server to the developers machine. For this data transfer, the developer can configure another set of options. As the data can be large, we offer a method of compressing the data during the transfer, leading to faster transfer times. In addition, the developer can choose to execute the UDF using a uniform random sample of the input data instead of the full set of input data. This will alleviate the data transfer overhead.

Since the data contained inside the database server might be sensitive, and it must be exported for debugging purposes, we also offer an optional encryption feature that can be used to safely transfer the sensitive data.

7.3 Implementation

The devUDF plugin works by connecting to the database using a JDBC connection. It then extracts the source code of the UDF together with its input parameters from the database by querying the databases' meta tables. An example of how MonetDB stores the source code of a Python function is shown in Listing 4.6. In order to be able to execute the UDF locally a set of code transformations has to be applied to this code, as the database only contains the function body. We need to create the header of the function using the function name and its parameters. To then run the created function, we need to obtain the input data from the database. In the generated code, we load the input data from a binary blob using the `pickle` library and pass it as a parameter to the function. When the user wants to export the UDF back to the database, these transformations are reversed and only the function body is committed.

When the user wants to debug the UDF locally using the interactive debugger, the input data of the function has to be extracted from the database. To obtain the input data, we take the user-submitted SQL query containing the call to the UDF,

and we replace the call to the UDF with a predefined extract function that transfers the input data back to the client instead of executing the UDF inside the server. We then run the transformed SQL query inside the database server to obtain the input data, store it on the developers machine and run the code of the transformed UDF.

The extract function used changes depending on the data transfer options selected by the user. If encryption is requested, the data is encrypted by the extract function before being transferred using the password of the database user as a key. The client then reverses the encryption to obtain the actual input data. The compression option works in a similar fashion. If the sample option is enabled, a uniform random sample of a size specified by the user is taken before extracting the data from the database server.

```
+----------------+-----------------------------------+
| name           | func                              |
+================+===================================+
| train_rnforest | {                                 |
:                : import pickle                     :
:                : from sklearn.ensemble             :
:                :     import RandomForestClassifier :
:                :                                   :
:                : clf = RandomForestClassifier(n)   :
:                : clf.fit(data, classes)            :
:                : return {'clf': pickle.dumps(clf), :
:                :      'estimators':n }             :
:                : };                                :
+----------------+-----------------------------------+
```

Listing 4.6: MonetDB UDF example.

8 Summary

In this chapter, we have introduced the vectorized MonetDB/Python UDFs. As both MonetDB and the vector-based language Python share the same efficient data representation, we can convert the data between the two separate formats in constant

time, as only the metadata has to be converted. In addition, as MonetDB operates on data in an operator-at-a-time fashion, no additional overhead is incurred for executing the UDFs in a vector-based fashion.

We have shown that MonetDB/Python UDFs are as efficient as UDFs written in the databases' native language, but without any of the downsides. MonetDB/Python UDFs can be created without requiring in-depth knowledge of the database kernel, and without having to compile and link the functions to the database server.

In addition, MonetDB/Python functions support automatic parallelization of functions over the cores of a single node, allowing for highly efficient computation. MonetDB/Python functions can be nested together to create relational chains, and parallel MonetDB/Python functions can be nested to perform Map/Reduce type jobs. All these factors make MonetDB/Python functions highly suitable for efficient in-database analysis.