Vectorized UDFs in Column-Stores

Mark Raasveldt
CWI
Amsterdam, the Netherlands
m.raasveldt@cwi.nl

Hannes Mühleisen
CWI
Amsterdam, the Netherlands
hannes@cwi.nl

ABSTRACT

Data Scientists rely on vector-based scripting languages such as R, Python and MATLAB to perform ad-hoc data analysis on potentially large data sets. When facing large data sets, they are only efficient when data is processed using vectorized or bulk operations. At the same time, overwhelming volume and variety of data as well as parsing overhead suggests that the use of specialized analytical data management systems would be beneficial. Data might also already be stored in a database. Efficient execution of data analysis programs such as data mining directly inside a database greatly improves analysis efficiency.

We investigate how these vector-based languages can be efficiently integrated in the processing model of operator–at–a–time databases. We present MonetDB/Python, a new system that combines the open-source database MonetDB with the vector-based language Python. In our evaluation, we demonstrate efficiency gains of orders of magnitude.

CCS Concepts

•**Information systems** → *Database query processing; Relational parallel and distributed DBMSs; Query languages;*

Keywords

Databases, User-Defined Functions, Column-Stores, Operator–at–a–Time Processing

1. INTRODUCTION

Transferring large amounts of data from a data management system for analysis purposes quickly becomes inefficient. Instead of transferring the data to the application, complex operations can be performed directly in the database. This can significantly improve performance, as the raw data does not have to leave the database server. Instead, only the necessary results are shipped back to the client.

However, most data analysis, data mining and classification operators are difficult and inefficient to express in SQL.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SSDBM '16 Budapest, Hungary

© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4215-5/16/07…$15.00

DOI: http://dx.doi.org/10.1145/2949689.2949703

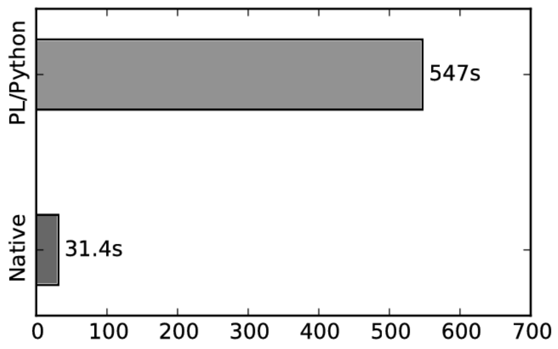


Figure 1: Modulo computation in Postgres.

The SQL standard describes a number of built-in scalar functions and aggregates, such as *AVG* and *SUM* [9]. However, this small number of functions and aggregates is not sufficient to perform complex data analysis tasks [17].

Database vendors often ship their own set of functions in addition to the functions defined by the standard. However, there are such a large number of specialized functions and aggregates used by data scientists that adding all possible functions required by them is infeasible [17].

Instead of using standard SQL queries, application logic can be executed in the database server through the use of *stored procedures*. Many database vendors provide procedural (turing-complete) extensions to SQL, such as Oracle's *PL/SQL* and Postgres' *PL/pgSQL*. However, many scientific and analytical algorithms are too complex to be implemented in SQL, even with these procedural extensions.

In addition to stored procedures, the SQL standard defines *external routines* or *user-defined functions*. These are stored routines implemented in different programming languages. Typically, databases support user-defined functions in compiled languages, such as C, C++ or JAVA. While user-defined functions written in these languages are efficient and flexible, they require the user to have in-depth knowledge of the specific database kernel they are written for, and as such are not easily portable to other databases or applications.

Many data scientists also do not use compiled languages such as C or Java to perform analytical data processing. Instead, they rely on interpreted languages such as R, Python or MATLAB [12] as there are large bodies of existing code that allow researchers to easily perform data science tasks.

Making user-defined functions available in these interpreted languages solves the above issues. By converting the data from the unique representation used by the database to a standard representation in an interpreted language, the user does not require in-depth knowledge of the database kernel to write these functions. And as these languages are interpreted, they do not need to be compiled and linked against the database server.

However, there are several issues that must be solved to efficiently use these vector-based languages as 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.

This issue is demonstrated in Figure 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.

Instead, 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.

In this paper, we investigate how vector-based languages can be efficiently integrated into the processing model of a database. We present our system, MonetDB/Python. In this system, we combine the open-source database system MonetDB with the interpreted language Python in an efficient way. We show how Python UDFs can be as efficient as UDFs in compiled languages, without any of the pitfalls that make compiled language UDFs hard to use.

MonetDB/Python is open-source. The source code is freely available online in the official MonetDB source code repository [1].

Contributions. The main contributions of this paper are as follows.

- We demonstrate that naively executing vector-based languages in a non-vectorized fashion is detrimental to performance.
- 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 column-store operator–at–a–time 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.

Outline. The paper is organized as follows. In Section 2, we review different types of user-defined functions and the influence of database architectures on user-defined functions. In Section 3, we present a brief background of both systems used. In Section 4, we present our system, MonetDB/Python.In Section 5, 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 6, we present related work. In Section 7, we describe how our work could be applied to other databases. Finally, in Section 8, we draw conclusions and discuss future work.

2. BACKGROUND

While all user-defined functions avoid unnecessary data transfer from the database to the client, the flexibility, ease of use and performance of these user-defined functions varies greatly. In this section we will give a brief overview of the different types of user-defined functions. In addition, we will give an overview of various different database architectures, and how these differences influence the performance and capabilities of user-defined functions.

2.1 Types of User-Defined Functions

User-defined functions can be used for a variety of different purposes, but what all user-defined functions have in common is that they interact with the data in the database in some fashion. They take data from the database as input, process that data and then output the processed data. The processed data can then be stored in the database or used in subsequent queries. UDFs differ in how many input values they can process at a time, how many input columns they can handle, and how many columns or rows they can output.

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 used in the *SELECT* and *WHERE* statements of a SQL query. An example of a scalar user-defined function is the multiplication of two columns $i * j$.

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. An example of an aggregate user-defined function is the `MAX` function, that returns the maximum of all the values in a column.

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

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. The possible input of table creating functions vary depending on the database. Certain databases only support the input of scalar values, whereas others support the input of an arbitrary amount of columns.

2.2 Physical Database Storage

The physical layout of the database influences the way in which the database can load and process data, and can significantly influence the performance of the database system when the user-defined function only uses a small amount of the input columns. The different physical layouts of database systems are shown in Figure 2.

Row Storage Databases fragment tables horizontally. In this storage model, the data of a single tuple is tightly packed. The main advantage of this approach is that updates to individual tuples are very efficient, as the data for a single tuple is tightly packed at a single location. However, columns cannot be loaded individually from disk because the values of a single column are surrounded by the values of the other columns.

As row-stores can only load entire tables, unused columns will affect query performance. When a query only operates on a subset of the columns of a table, the entire table must be loaded. This is especially relevant for analytical functions that only touch a few columns in large tables with hundreds or even thousands of columns.

Column Storage Databases fragment tables vertically. In this storage model, the data of the individual columns is tightly packed. The main advantage of this approach is that columns can be loaded and used individually, which means we do not need to load in any unused columns. However, operations on individual tuples are inefficient because they are spread across the different columns.

2.3 Database Processing Model

The processing model of the database heavily influences the design and performance of the user-defined functions, as the processing model defines how the data is transferred between the database and the user-defined function. The processing model is closely related to the physical storage of the database.

Tuple-at-a-Time Processing is the standard processing model used by most row-oriented databases. In this processing model, the individual rows of the database are processed one by one from the start of the query to the end of the query.

User-defined functions in tuple–at–a–time databases typically follow this processing model. As input, they receive a single row and process this row, then output the processed row again.

This approach lends itself well to user-defined scalar functions, but has to be adapted for user-defined aggregate functions. As aggregate functions depend on the values of the entire column, rather than only on the values of a single row.

User-defined aggregate functions have to either be computed incrementally, or the data from the relevant columns has to be gathered at a single location. Incremental computation is only possible for simple aggregates such as the `MAX` function. For more complex aggregates, such as the `MEDIAN`, the database must first iterate over the data and copy the relevant columns to a separate storage location before computing the aggregate.

The main disadvantage of the tuple–at–a–time processing model is the function call overhead incurred for computing the individual rows. Every step in the query execution requires one function call for every tuple. This overhead is significant even in compiled languages. For vector-based languages, this overhead is so large that it makes processing large amounts of data an impossibility.

Pipeline Processing is the processing model used by the hybrid OLTP/OLAP database HyPeR [15]. The pipeline processing model is similar to the tuple–at–a–time processing model, in that it maximizes data locality by processing one tuple at a time.

However, unlike standard tuple–at–a–time processing models, it avoids the per-tuple function call overhead by generating code. While this adds additional compilation overhead to queries, the reduced function-call overhead results in faster query execution time.

Operator–at–a–Time Processing is an alternative query processing model. Instead of processing the individual tuples one by one, the individual operators of the query are executed on the entire columns in order. As the operators process entire columns at a time, the function call overhead of this processing model is minimal.

In the operator–at–a–time processing model, user-defined functions are called once with all the data is input, rather than n times with a single row as input.

In this processing model there is no difference in how user-defined scalar, aggregate and table functions are processed. As user-defined functions always have access to the entire input columns, there is no need to distinguish between functions that require access to all the data, and functions that only operate on part of the data.

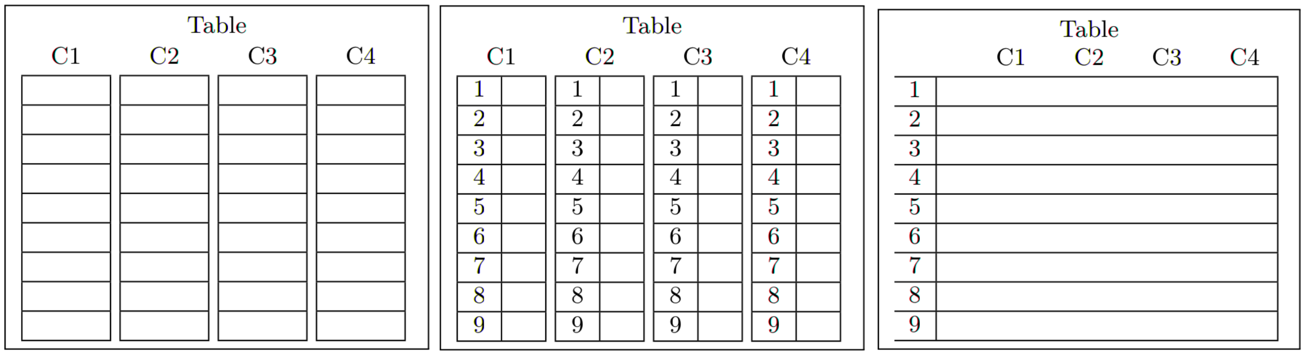
The main drawback of this processing model is the materialization cost of the intermediates of the operators. In the tuple-at-a-time processing model, a single tuple is processed from start to finish before the query processor moves on to the next tuple. By contrast, in the operator–at–a–time processing model, the operator processes the entire column at once before moving on to the next operator. Because of this, the intermediate result of every operator has to be materialized into memory so the result can be used by the next operator. As these intermediate results are the result of an entire column being processed they can take up a significant amount of memory if they are not compressed.

Vectorized Processing is a hybrid processing model that sits between the *tuple-at-a-time* and the *operator–at–a–time* models. It avoids high materialization costs by operating on smaller chunks of rows at a time, while also avoiding overhead from a significant amount of function calls. This approach is used by Vectorwise [4] and Vertica [13].

User-defined functions in this processing model are similar to user-defined functions in the *tuple–at–a–time* processing model. User-defined scalar functions fit directly into this processing model. However, aggregate functions that need access to the entire column force the database to copy and gather the input columns to a single location.

3. COMPONENTS

MonetDB/Python combines the relational database MonetDB with the interpreted language Python. In this section, we will provide a brief background on both these systems.



(a) Column-Store with virtual identifiers. (b) Column-Store with explicit identifiers. (c) Row-Store.

Figure 2: Physical layout of column-store and row-store databases.

3.1 MonetDB

MonetDB is an open source column-store RDBMS that is designed primarily for data warehouse applications. In these scenarios, there are frequent analytical queries on the database, often involving only a subset of the columns of the tables, and unlike typical transactional workloads, insertions and updates to the database are infrequent and in bulk or do not occur at all.

MonetDB is tuned for these analytical workloads by vertically fragmenting the data. This design allows operations within the database to operate on individual columns.

Every relational table is stored as a set of Binary Association Tables (BATs). Every column in the table is represented by a single BAT. The BAT stores the column as a set of *oid-value* pairs, where the *oid* (object-id) describes which row the value belongs to, and the *value* contains the actual value for the field.

The *oid* of each value is not stored explicitly in MonetDB, but rather implicitly, as shown in Figure 2a. This is accomplished by only storing the *oid* of the first element, oid_{base}, and ensuring that the subsequent values have incremental *oids*. The *oid* of the i^{th} value is then $oid_{base} + i$.

The physical storage model is not the only way in which MonetDB is optimized for analytical queries. The entire execution model of the database is designed around late tuple reconstruction [8]. MonetDB processes the data in an operator–at–a–time manner, and only reconstructs the tuples just before sending the final result to the client. This approach allows the query engine to use vectorized operators that process entire columns at a time.

3.2 Python

Python is a popular interpreted language, that is widely used by data scientists. It is easily extensible through the use of modules. There are a wide variety of modules available for common data science tasks, such as `numpy`, `scipy`, `sympy`, `sklearn`, `pandas` and `matplotlib`. These modules offer functions for high performance data analytics, data mining, classification, machine learning and plotting.

While there are various Python interpreters, the most commonly used interpreter is the CPython interpreter. This interpreter is written in the language *C*, and provides bindings that allow users to extend Python with modules written in *C*.

Internally, CPython stores every variable as a `PyObject`. In addition to the value this object holds, such as an integer or a string, this object holds type information and a reference count. As every `PyObject` can be individually deleted by the garbage collector, every Python object has to be individually allocated on the heap.

The internal design of CPython has several performance implications that make it unsuitable for working with large amounts of data. As every `PyObject` holds a reference count (64-bit integer) and type information (pointer), every object has 16 bytes of overhead on 64-bit systems. This means that a single 4-byte integer requires 20 bytes of storage. In addition, as every `PyObject` has to be individually allocated on the heap, constructing a large amount of individual Python objects is very expensive.

Instead of storing every individual value as a Python object, packages intended for processing large amounts of data work with NumPy arrays instead. Rather than storing a single value as a `PyObject`, a NumPy array is a single `PyObject` that stores an array of values. This makes this overhead less significant, as the overhead is only incurred once for every array rather than once for every value.

This solves the storage issue, but standard Python functions can only operate on `PyObjects`. Thus if we want to actually operate on the individual values in Python, we would still have to convert each individual value to a `PyObject`.

The solution employed in Python (and other vector-based languages) is to have *vectorized functions* that directly operate on all the data in an array. By using these functions, the individual values are never loaded into Python. Instead, these vectorized operations are written in *C* and operates directly on the underlying array. As these functions operate on large chunks of data at the same time they also make liberal use of *SIMD* instructions, allowing these vectorized functions to be as fast as optimized *C* implementations.

4. 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.

4.1 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 1.

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

Listing 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 2: Simple Scalar UDF.

An example of a scalar function that computes the square root of a set of integers is given in Listing 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.

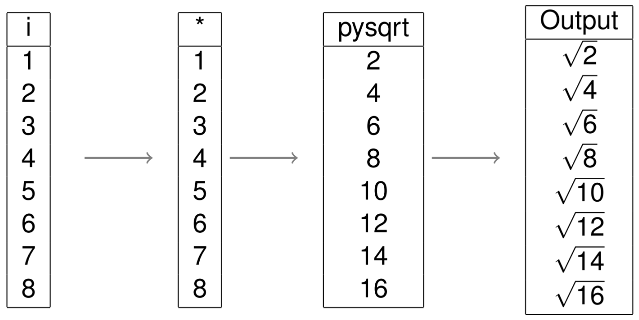


Figure 3: Operator Chain for Listing 2.

4.2 Processing Pipeline

MonetDB/Python functions are executed as an operator in the processing model of the database, as illustrated in Figure 3. MonetDB/Python functions run in the same process and memory space as the database server. As such, MonetDB/Python 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 4.3, 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 6.

4.3 Parallel Processing

In Section 4.2 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 [18]. 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.

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

Listing 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.

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.

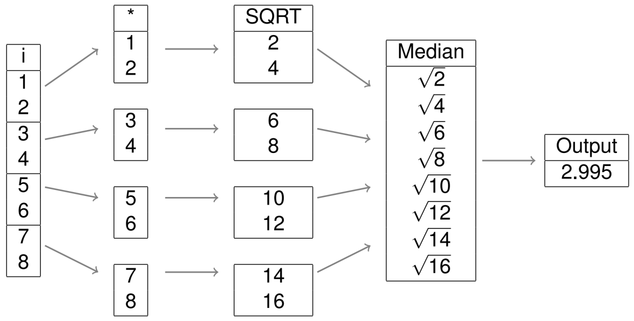


Figure 4: Parallel Operator Chain of Listing 3.

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.

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.

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.

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

Listing 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 5.

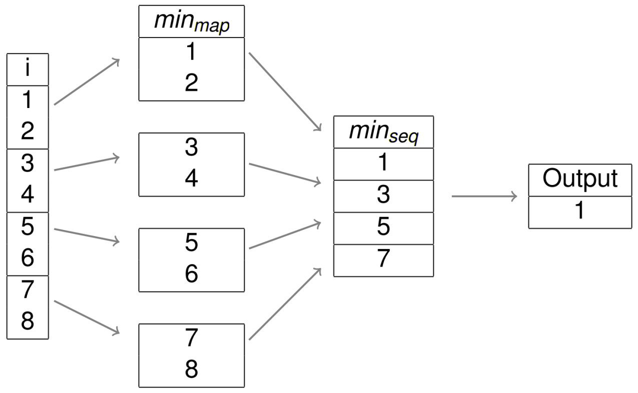


Figure 5: Operator Chain of Listing 4.

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.

5. 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.

5.1 Systems Measured

MySQL is the most popular open-source relational database 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++* [1].

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 [3].

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* [2], 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.

5.2 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 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.

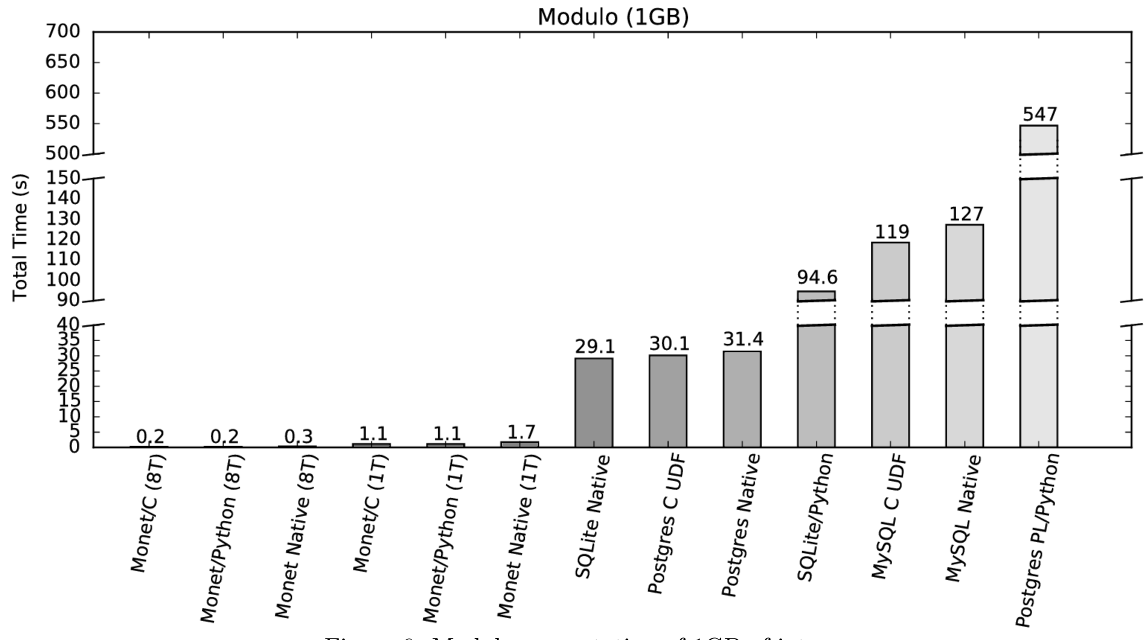


Figure 6: Modulo computation of 1GB of integers.

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 6. 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.

6. 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.

6.1 Research

Research on user-defined functions started long before they were introduced into the SQL standard. The work by Linnemann et al. [14] 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.

6.1.1 *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. [5, 6] 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. [11]. 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. [5] and Jaedicke et al. [11] 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 [16] 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)*.

6.1.2 *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.

6.1.3 *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. [10] 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.

6.2 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.

6.2.1 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* [7]. 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. [10], 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.

7. 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.

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.

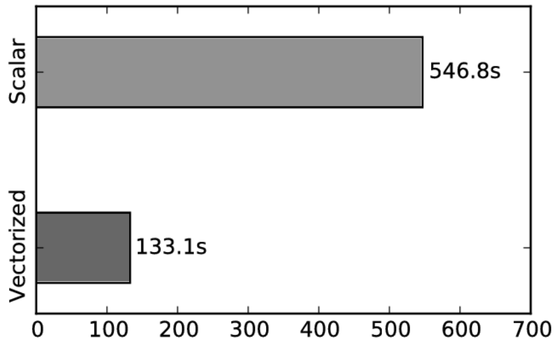


Figure 7: PL/Python Vectorized vs Non-Vectorized Modulo Operator.

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 7. 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.

Pipeline Processing faces the same issues as tuple-at-a-time processing in this scenario. Pipeline processing avoids function call overhead by generating code. The only way to avoid this function call overhead for UDFs written in vector-based languages would be to translate the code written in the vector-based language to the databases' native language. That translated code can then be inlined into the generated code, which would avoid function call and interpreter overhead. While this might be possible for simple functions, it is not feasible for arbitrary functions.

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, as the vector-based language needs access to the entire input columns.

8. CONCLUSION

In this paper, 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.

8.1 Future Work

While MonetDB/Python functions are already very usable for efficient in-database analytics, there are still improvements that can be made to the system.

Polymorphism. Currently, MonetDB/Python functions are only partially polymorphic. The user can specify that the function accepts an arbitrary number of arguments, however, the return types are still fixed and must be specified when the function is created. Allowing the user to create complete polymorphic functions would increase the flexibility of MonetDB/Python functions.

The problem with polymorphic return types is that the return types of the function must be known while constructing the query plan in the current execution engine. Thus we cannot execute the function and look at the returned values to determine the column types. The solution proposed by Friedman et al. [7] is to allow the user to create a function that specifies the output columns of the function based on the types of input columns. This function is then called while constructing the query plan to determine the output types of the function.

This allows the user to create functions whose output columns depend on the number of input columns and the types of those columns. However, it does not allow the user to vary the output columns based on the actual data within the input columns. Consider, for example, a function that takes as input a set of JSON encoded objects, and converts these objects to a set of database columns. The amount of output columns depends on the actual data within the JSON encoded objects, and not on the amount or type of the input columns, thus these types of polymorphic user-defined functions are not possible using the proposed solution.

The ideal solution would be to determine the amount of columns during query execution, however, this provides several challenges as the query plan must be adapted to the amount of columns returned by the function, and must thus be dynamically modified during execution.

Data Partitioning. MonetDB/Python supports parallel execution of user-defined functions. It does so by partitioning the input columns and executing the function on each of the partitions. Currently, the partitioning simply splits the input columns into n equally sized pieces. This is the most efficient way of splitting the columns, but it limits the parallelizability of user-defined functions. Functions that operate only on the individual rows, such as word count, can be parallelized using this partitioning.

However, as noted by Jaedicke et al. [10], certain functions cannot be efficiently executed in parallel on arbitrary partitions, but can be efficiently computed in parallel if there are certain restrictions on the partitioning scheme. Allowing the user to specify a specific partitioning scheme would increase the flexibility of the parallelization.

There are performance implications in arbitrary partitioning in a column-store. Normally, the identifiers of every row are not explicitly stored, as shown in Figure 2a. The current partitioning scheme does not rearrange the values in the columns, which allows these identifiers to remain virtual. However, if we rearrange the values in the columns to match a user-defined partitioning scheme, we would need to explicitly store the row identifiers, resulting in significant additional overhead. This is avoided by the special partitioning used for computing parallel aggregates, because we do not need to know the individual tuple identifiers of each of the values as we are accumulating the actual values, thus we only need to know the group that the value belongs to.

Still, parallelization could lead to big improvements in execution time of CPU-bound functions. It would be interesting to see how big the set of functions is that cannot be parallelized over arbitrary partitions, but can be parallelized over restricted partitions. It would also be interesting to see if it would be worth the performance hit of creating these restricted partitions over the data so we can compute these functions in parallel.

Distributed Execution. Currently, MonetDB/Python can only be parallelized over the cores of a single machine. While this is suitable for a lot of use cases, certain data sets cannot fit on a single node and must be scaled to a cluster of machines. It would be interesting to scale MonetDB/Python functions to work across a cluster of machines, and examine the performance challenges in a parallel database environment.

Script Optimization. In this paper we have focused mainly on optimizing the dataflow around user-defined functions. We have seen in Figure 6 that this dramatically speeds up functions for which transportation of data is the main bottleneck. However, when the computation time dominates the transportation time this optimization will not provide a significant speedup. We have provided the ability to execute functions in parallel, which can still provide significant speedups to these functions. However, we still treat the user-defined functions as black boxes. Additional speedups could be achieved by looking into the user-defined functions and optimizing the code within the functions.

Cardinality Estimation. MonetDB uses heuristics based on table size when creating the query plan to determine how the columns should be partitioned for parallelization, as partitioning small tables significantly degrades performance. However, when the table is generated by a table-producing function, this table could potentially have any size. An interesting research direction could be estimating the cardinality of these table-producing functions.

Code Translation. When creating MonetDB/Python, we have tried to make it as easy as possible for data scientists to make and use user-defined functions. However, they still have to write user-defined functions and use SQL queries to use them if they want to execute their code in the database. They would prefer to just write simple Python or R scripts and not have to deal with database interaction.

An interesting research direction could be analyzing these scripts, and automatically shipping parts of the script to be executed on the database as user-defined functions. This way, data scientists do not have to interact with the database at all, while still getting the benefits of user-defined functions.

Acknowledgments

This work was funded by the Netherlands Organisation for Scientific Research (NWO), project "Process mining for multi-objective online control" (Raasveldt) and "Capturing the Laws of Data Nature" (Mühleisen). We thank Sjoerd Mullender and Niels Nes of the MonetDB team for their help with the implementation.

9. REFERENCES

[1] Adding a new user-defined function. In *MySQL User Manual*.

[2] Create or redefine sql functions. In *SQLite User Manual*.

[3] Procedural languages. In *PostgreSQL User Manual*.

[4] P. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR*, 2005.

[5] Q. Chen, M. Hsu, and R. Liu. Extend udf technology for integrated analytics. In T. Pedersen, M. Mohania, and A. Tjoa, editors, *Data Warehousing and Knowledge Discovery*, volume 5691 of *Lecture Notes in Computer Science*, pages 256–270. Springer Berlin Heidelberg, 2009.

[6] Q. Chen, M. Hsu, R. Liu, and W. Wang. Scaling-up and speeding-up video analytics inside database engine. In S. Bhowmick, J. KÃijng, and R. Wagner, editors, *Database and Expert Systems Applications*, volume 5690 of *Lecture Notes in Computer Science*, pages 244–254. Springer Berlin Heidelberg, 2009.

[7] E. Friedman, P. Pawlowski, and J. Cieslewicz. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.*, 2(2):1402–1413, Aug. 2009.

[8] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull*, page 2012.

[9] ISO. Iso/iec 9075:1992, database language sql. Technical report, July 1992.

[10] M. Jaedicke and B. Mitschang. On parallel processing of aggregate and scalar functions in object-relational dbms. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, pages 379–389, New York, NY, USA, 1998. ACM.

[11] M. Jaedicke and B. Mitschang. User-defined table operators: Enhancing extensibility for ordbms. In M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 494–505. Morgan Kaufmann, 1999.

[12] R. M. John King. 2015 data science salary survey. Sept. 2015.

[13] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, Aug. 2012.

[14] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 294–305, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[15] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[16] C.-F. Sundlöf. In-database computations. Master's thesis, Royal Institute of Technology, Sweden, 10 2010.

[17] H. Wang and C. Zaniolo. User-defined aggregates in database languages. In R. Connor and A. Mendelzon, editors, *Research Issues in Structured and Semistructured Database Programming*, volume 1949 of *Lecture Notes in Computer Science*, pages 43–60. Springer Berlin Heidelberg, 2000.

[18] F. Wolf, I. Psaroudakis, N. May, A. Ailamaki, and K.-U. Sattler. Extending database task schedulers for multi-threaded application code. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 25:1–25:12, New York, NY, USA, 2015. ACM.