Vectorized UDFs in Column-Stores
Utrecht University

Mark Raasveldt

December 16, 2015

Contents

Abstract

An ever increasing amount of data is gathered by companies, government entities and individuals. Analysing and making sense of that data is a crucial task. A task that is getting more important as there is a shift towards more data-driven decision making in society.

The tools used by data scientists for ad-hoc data analysis are scripting languages, of which R and Python are the most popular. Scripting languages are flexible, easy to use and have a large existing code base for data analytics.

Relational Database Management Systems (RDBMS) are the de-facto standard for storing tabular data. RDBMS have numerous advantages when storing tabular data, amongst which are ACID properties, scalability, data validation and automatic parallelization.

If the user wants to use data stored in a RDBMS in a scripting language, the data has to be transferred from the RDBMS to the scripting language. The standard solution is a loosely coupled approach between the scripting language and the database using an ODBC connector. To transfer the data to the scripting language, the data is exported from the database and copied, often over a network connection, and converted between the differing formats of the database and the scripting language.

This loose-coupling approach has significant performance implications, especially when transferring data over the network. In addition, the lack of a tight integration has lead to data management features being re-implemented from scratch within scripting languages, by libraries such as Pandas and Dplyr.

The main contribution of this thesis is research towards how a scripting language can be tightly integrated into a columnar data management system. We present a new system, *MonetDB/Python*, which deeply integrates the Python scripting language into MonetDB, an open-source relational column store.

By using this system, users can execute arbitrary python functions as part of relational SQL queries inside the database process. This significantly reduces the cost of data transfer, and allows for automatic parallelization of scripting language functions.

We show that our method is not only faster than current RDBMS connectors, but also that it is faster than native storage solutions in Python. MonetDB/Python allows us to combine the scalability and power of a RDBMS with the flexibility of a scripting language without the drawback of slow transfer speed.

Chapter 1

Introduction

Relational Databases are among the most mature and most popular methods of storing data. RDBMS have numerous advantages, such as ACID[1] properties, scalability, availability, data validation and automatic parallelization. Relational Databases are excellent tools for storing relational data, as they enable flexible retrieval of the data through the use of joins and avoid data redundancy by storing data in separate tables and joining them together when needed. They are based on the logical relational model [1].

Most relational databases use the SQL query language to allow users to retrieve data and perform various operations on the data in the database [2]. SQL allows the user to perform complex data retrieval and data manipulation without any knowledge of the underlying physical structure of the data.

However, most data analysis, data mining and classification operators are prohibitively difficult and inefficient to express in current SQL-compliant database systems. The SQL standard describes a number of built-in scalar functions and aggregates, such as *AVG* and *SUM* [3]. However, this small number of functions and aggregates is not sufficient to perform complex data analysis tasks [4].

Most database vendors 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 in data mining, data analysis, classification and machine learning that adding all possible functions required by users is infeasible [4]. In addition, users could devise their own customized functions and algorithms to perform these tasks.

Instead of SQL, data scientists use procedural languages to perform more complex operations on data. Scripting languages in particular are popular amongst data scientists [5]. Scripting languages are highly extendable, have a large existing code base for numerous mathematical and statistical operations and have a low barrier of entry. These advantages make scripting languages ideal for data analysis, data mining, machine learning, classification, natural language processing, graph plotting and mathematics.

As the name implies, data scientists work with data in some way. Thus one of the core problems data scientists have to deal with is finding a suitable place to store the data. When dealing with a small volume of data, data can be stored in portable CSV files or other flat file storage solutions. However, flat file storage does not scale well. Portable files such as CSV files are encoded in a non-binary

[1]Atomicity, Consistency, Isolation, Durability

format, which means reading the data from them is inefficient. We must first parse the file and convert the contents to the binary types in the scripting language. This is acceptable for smaller data sets, but the load times can grow out of control for larger data sets. In addition, managing individual files manually quickly becomes unwieldy when dealing with data sets that are more complex than a few separate tables.

As we have argued, RDBMS are an excellent storage place for relational data and are much more scalable than flat files, but to use them in this context we need some way of combining procedural languages with RDBMS. There are two solutions for this situation.

The most common solution that is employed by most vendors is loose coupling of the procedural language with the RDBMS through an ODBC connector [6]. However, this loose-coupling approach has significant performance implications [7] as the data must be exported from the database.

The alternative solution is tight coupling of the procedural language with the RDBMS by allowing the user to create *user-defined functions* (UDFs). These user-defined functions can then be used within SQL statements to perform operations that are too slow or too complex to express in SQL itself.

However, most database vendors only have limited support for user-defined functions in procedural languages. Many popular RDBMS, such as MySQL [8] and Oracle [9], only have support for user-defined functions in compiled languages such as *C*. It is difficult to perform complex data analysis in these languages, and creating these user-defined functions often requires extensive knowledge of the underlying structure of the database kernel.

The databases that do implement support for scripting languages, such as Postgres [10], Vertica [11] and AsterData [12], use the *tuple-at-a-time* processing method, which introduces a significant amount of overhead when transferring the data to scripting languages in a vectorized format.

Purpose of this Thesis

In this thesis, we propose a new way of combining columnar relational database management systems and scripting languages. By allowing users to execute scripting language functions directly within the SQL layer as SQL user-defined functions (UDFs), users can use the scripting language to execute complex data analysis operations, while using SQL for its intended purpose: data management.

As the UDFs are executed in the database server, which is also where the data resides, this method avoids the significant transfer overhead that is incurred when the data is transferred over an ODBC connection. In addition, by executing the scripting language functions within the SQL layer, we can take advantage of the automatic parallelization offered by the database to execute UDFs in parallel.

By executing the user-defined functions in a database that uses *operator-at-a-time* processing, we can efficiently transfer the data to the scripting language in a vectorized format. Transferring the data in a vectorized format allows the user to take advantage of the efficient vectorized processing present in popular scripting languages. This is much more efficient as it allows them to avoid the per-element interpreter overhead that makes loops in scripting languages inefficient.

For the implementation, we use Python as the scripting language. Python is one of the most popular scripting languages. It is a general purpose scripting language that is widely used by scientists and software engineers. It is one of the most popular languages in data science [13] and is widely taught in academic institutions [14]. It has a large existing code base for data analysis, data mining, machine learning, natural language processing, classification, graph plotting and mathematics. Most of these libraries are optimized for efficient vectorized processing. These combined factors make Python an ideal scripting language for performing efficient operations on a database, while maintaining the advantages of a scripting language.

As we are mainly focusing on analytical workloads, a column store database offers many advantages over a row-store database [15, 16]. For classification and data analytics, we typically do not need access to the entire table all the time. The column store model ensures that we do not spend time reading irrelevant columns into memory.

For the implementation, we use MonetDB [17] as the RDBMS. MonetDB is an open source column-oriented DBMS that uses the *operator-at-a-time* processing model. MonetDB is actively used in health care, telecommunications and astronomy. It is a mature database that has existed since 1993 [18]. An in-depth explanation of how MonetDB works internally is given in Chapter 3.1.

Thesis Outline

The rest of the thesis is structured as follows. In Chapter 2, we discuss the related research that has been done on user-defined functions, and discuss the support of user-defined functions in popular RDBMS vendors. In Chapter 3.1, we provide a detailed view of the MonetDB kernel. In Chapter 3.2, we discuss the internal workings of Python, and how this affects the performance of user-defined functions. In Chapter 4, we provide an in-depth explanation of our implementation, and various choices we have made during the implementation. In Chapter 5, we perform various benchmarks to test the performance of our user-defined functions compared to alternative storage solutions and other UDF implementations. Finally, in Chapter 6, we draw our conclusions and present future work.

In Appendix A, the source code used for the microbenchmarks performed throughout the thesis is shown. In Appendix B, we discuss some additional challenges we encountered when converting data from MonetDB to Python that were too detailed to include in the main thesis. In Appendix C, we introduce loopback queries, a feature that allows users to query the database from within the user-defined function. In Appendix D, we use MonetDB/Python to perform actual data science, in the form of classification, and show that it is not only fast when used with small micro benchmarks but is also highly suitable for actual data science.

Chapter 2

Related Work

User-defined functions are well established database features that allow users to extend database functionality. They allow user customization of how a database processes data. There has been significant research related to user-defined functions. User-defined functions were introduced into the SQL standard in SQL:1999 [19].

In this section, we will discuss the different types of user-defined functions, give an overview of the research that has been done on user-defined functions, and give a brief overview of various implementations of user-defined functions by popular RDBMS vendors.

2.1 User-Defined Functions

In this section, we will give a brief overview of the different types of user-defined functions and their characteristics. We differentiate them based on the amount of data the user-defined function processes, how the data is transferred to the user-defined function and the language in which it is written. Furthermore, the database processing model and the physical storage of the database that the user-defined function resides in also significantly influence its performance.

Data Processing

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, and then output data that can be used in the database. They differ in how they interact with the input columns and what type of output they generate.

User-Defined Scalar Functions are *n-to-n* operations that operate only on the individual rows of the input columns. An example of a scalar user-defined function is the multiplication of two columns $i * j$. These user-defined functions directly fit into the tuple-at-a-time processing model and can be automatically parallelized by executing the user-defined function in parallel on different sections of the input columns.

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. These user-defined functions do not directly fit into the tuple-at-a-time processing model. They have to be either computed incrementally, which is only possible for certain aggregates such as the `MAX` function, or they require the data to be copied to a separate column so the user-defined function can have access to the entire column at the same time.

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.

Data Transfer

As user-defined functions operate on the data in the database in some fashion, they naturally need access to that data, thus the data needs to be transferred to the user-defined function in some way.

Code-Shipping means that the source code of the function and any static data necessary for execution is transferred from the client to the server. The function itself is executed in the database server. As the data also resides on the database server, the data does not have to be transferred to the client.

Data-Shipping means that the user-defined function is executed on the machine of the client. As the data still resides on the database server, the data must be transferred (or shipped) to the client. This occurs when the user uses an ODBC connection to the database to load the data, and then executes the function on the loaded data. Research has also been done on executing SQL UDFs on the clients machine [20]. The main advantage of this approach is that it reduces security issues introduced by allowing users to execute arbitrary functions directly on the database server.

Programming Language

The user-defined function must be written in some programming language, we differentiate between different groups of programming languages.

SQL is the standard querying language in most DBMS. Allowing users to write user-defined functions in SQL is a natural extension for most databases. However, SQL is not a procedural language. As a result of this, performing complex operations is very difficult in SQL. The solution employed by most vendors is to have a number of vendor-specific extensions for SQL that allow it to function as a procedural language. Examples of these UDFs are Oracle's *PL/SQL* and Postgres' *PL/pgSQL*.

Compiled Languages are languages that have to be manually compiled and linked to the database before they can be used as SQL functions. The most common compiled language available for use in user-defined functions is *C*. These user-defined functions are very efficient, but they are difficult to use and often require the user to have extensive knowledge about the database kernel. They are especially difficult to use for complex data science operations.

Scripting Languages are interpreted languages, and as such only require the actual source code to be executed. These functions do not need to be compiled or linked to the database, but can be used immediately after shipping the source code to the database server.

Database Processing Model

The processing model of the database heavily influences the performance of the user-defined functions, as the processing model defines how the data is transferred to the user-defined function.

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. This approach lends itself well to scalar user-defined functions, as the scalar function directly fits into the processing model and can be executed once for every tuple. However, even though these scalar functions fit naturally into this processing model there is still significant function call overhead from calling the function once for every tuple, especially when the function has to be computed for a large amount of rows. This is especially true when the user-defined function is written in a scripting language that has to be interpreted, or for languages that are not compiled to machine code such as Java.

The *tuple-at-a-time* processing model is even more problematic when the user-defined function needs access to an entire column to perform an aggregation. In this case, the function cannot be executed as part of the query flow. Instead, the database has to scan the table, create a copy of all the relevant tuples in a separate column and then call the user-defined function with the complete copy of the column, which is a very expensive operation.

Operator-at-a-Time Processing is an alternative query processing model that is used by MonetDB. 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 work on entire columns at a time, there is significantly reduced function call overhead. The user-defined function is only called once, rather than once for every tuple. In addition, both the scalar and aggregate functions fit directly into the query flow.

The main drawback of this processing model is the materialization 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 [21] and Vertica [22].

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.

Row Storage Databases horizontally fragment tables, as illustrated in Figure 3.1c. In this storage model, the data of a single tuple is tightly packed together. 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, thus unused columns will affect query performance significantly.

Column Storage Databases vertically fragment tables, as illustrated in Figure 3.1b. In this storage model, the data of the individual columns is tightly packed together. 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 or unnecessary columns. However, operations on individual tuples are inefficient because they are spread across the different columns.

2.2 Research

Research on user-defined functions started long before they were introduced into the SQL standard. The work by Linnemann et al. [23] 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 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.

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. [24, 25] explores how *C* UDFs can be used to implement OLAP cubes in a database. The motivation behind this work is that traditional OLAP cube analysis was either done using external tools, in which case the data must be exported from the database, or using OLAP SQL queries, which are inefficient. The authors realized their implementation in the row-store database *SQL Server*, and found that their implementation was significantly more efficient than the alternative OLAP SQL queries.

The work by Ordonez [26] discusses how a set of statistical models, such as linear correlation and linear regression, can be implemented in a database using C UDFs. They implemented a set of statistical models in Teradata, a hybrid column-row store database, using three separate methods. They implemented the models in C UDFs, as a set of SQL queries and in $C++$ connected to the database using an ODBC connector. They found that user-defined functions in C outperformed both alternative solutions by a considerable margin.

The work by Chen et al. [27, 28] 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.

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. The same observation is made by Jaedicke et al. [29]. 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. [27] and Jaedicke et al. [29] propose a new set of user-defined functions that can take a relation as input and produce a relation as output.

The work by Sundlöf [30] 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.

UDF Data Flow

The order in which the database processes operations is central to optimizations in databases. The order in which joins and filtering functions are processed can have a significant performance impact, even though the final result of the query will remain the same. Organizing the data flow becomes significantly more challenging when user-defined functions are involved, as they can behave very differently to built-in functions.

The work by Hellerstein et al. [31] and Chaudhuri et al. [32] focuses on user-defined predicates in the SQL Where clause. They make the observation that traditionally, query optimizers execute filtering functions as early as possible, as built-in SQL predicates are generally cheap functions, and reducing the total amount of rows prior to performing joins leads to significant performance improvements. However, user-defined predicates can be very expensive, so much so that executing them as early as possible can be detrimental to performance, especially when the user-defined predicates are not very selective.

They describe how query optimizers can take expensive predicates into account by providing the *selectivity* and *per-tuple cost* of computing such a predicate. However, they require the creator of the user-defined function to provide these variables. This places the burden of optimization on the user, and requires the user to have in-depth knowledge about the efficiency of their own function.

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. [33] 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. The user can choose a partitioning scheme.

`ANY`, that indicates the way the data is partitioned does not matter, and any partitioning is a valid partitioning.

`EQUAL (column name)`, which partitions the data such that all rows with the same value in the specified column belong to the same partition.

`RANGE (column name)`, which partitions the data such that all rows with values in a specific range in a specified column belong to the same partition.

`UDF`, which allows the user to create their own user-defined function that divides the data into separate partitions.

By allowing the user to specify their own partitioning scheme, a significant amount of aggregates can be computed in parallel.

2.3 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* [12]. 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. [33], 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 main difference between MonetDB/Python and SQL/MR functions is that SQL/MR functions operate on the rows in a *tuple-at-a-time* fashion, processing each row individually. The user obtains the next row by calling the `advanceToNextRow` function, and outputs a row using the `emitRow` function.

When the user wants to have access to the entire column, rather than just the individual rows, the user must call the `advanceToNextRow` function once for every row, and must then copy the value of each of the rows to create a copy of the input column. This creates significant overhead. Not only do we have to copy the entire column, we have to make a function call for every row in the input.

This is especially relevant when working with scripting languages such as *R* and *Python*. These scripting languages only process the data efficiently when using vectorized operations, thus when using these languages the user will almost always want access to the entire columns.

By contrast, MonetDB processes the data in a *operator-at-a-time* fashion. By processing the data one operator at a time, the individual operators always have access to the entire input columns without having to copy the data, and without any additional function call overhead. This allows MonetDB/Python to transfer the data into scripting languages in a vectorized fashion much more efficiently.

Vertica

Vertica is a commercial column-store database focused on data analytics and data warehousing. It started as a commercial branch from the open-source column-oriented database C-Store. It offers in-database processing through *User-Defined*

Extensions [11]. User-Defined Extensions can be written in both compiled languages (C++ and Java) and scripting languages (R).

User-Defined Functions can be run in either fenced or unfenced mode. In fenced mode, the user-defined functions run directly in the database process, which could cause issues if they leak memory or access uninitialized memory. In unfenced mode, the user-defined functions run in a separate process, which shields the database server process from any potential side-effects of the function but has additional overhead, as the input and output must be transferred between the processes.

Vertica uses the vectorized processing model [22], which is a hybrid model that sits between the *tuple-at-a-time* processing models and the *operator-at-a-time* processing models. Instead of operating on single tuples, or operating on entire columns, it operates on chunks of the columns.

The issues with this processing model are similar to the issues with the *tuple-at-a-time* processing model. While this processing model cuts back on the amount of function calls that need to be performed if we want to perform operations on an entire column, we still need to concatenate the individual vectors together to recreate the input columns, which requires us to copy them.

MySQL

MySQL is the most popular open-source relational database system [34]. 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* and *C++* [8].

MySQL only supports user-defined scalar and aggregate functions, and has no support for user-defined table functions. This greatly limits the functionality that user-defined functions can have, as they cannot create new relations.

MySQL is a row-store database that uses the *tuple-at-a-time* processing model. The row-store storage model causes additional performance issues when user-defined functions are used for data analytics, as analytical queries typically only operate on a small subset of all the columns of a table. The row-store model does not allow the database to load individual columns. Instead, the database must always load the entire table, including all unused columns. This can cause significant performance issues when working with large tables.

Postgres

Postgres is the second most popular open-source relational database system [34]. It is a row store database that focuses on being SQL compliant and having a large feature set. Postgres supports user-defined functions in both compiled languages (C and Java) and scripting languages (PHP, Perl, Python, R and Ruby) [10].

Postgres is similar to MySQL. It is a row-store database that uses the *tuple-at-a-time* processing model. However, Postgres has more extensive support for user-defined functions, and supports user-defined scalar, aggregate and table functions.

In addition, user-defined functions in Postgres support *loopback queries*. This allows the user to query the database from within the user-defined function, similar to an ODBC connection. However, because the loopback query is executed within the database process, these queries are more efficient than regular ODBC connections.

Chapter 3

Tools

In this section, we will give an in-depth description of the tools we have used to implement MonetDB/Python.

3.1 MonetDB

MonetDB is a popular open source RDBMS that is actively used in telecommunications, health care, astronomy, data management research and education [17]. It is a DBMS 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 or do not occur at all.

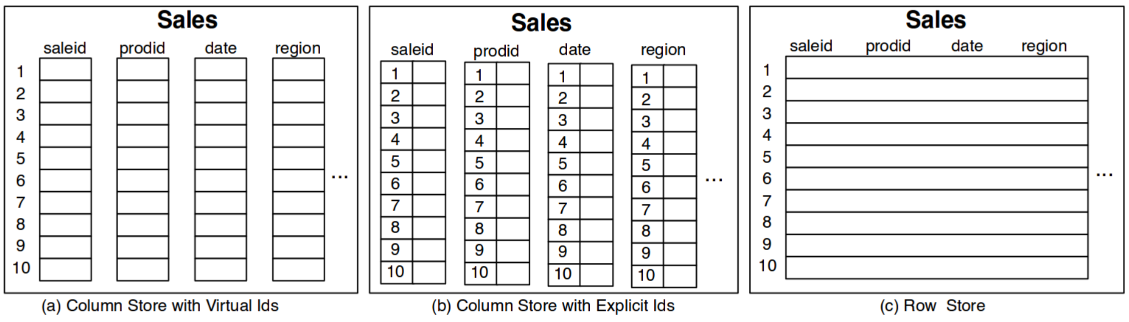

Sales
saleid
prodid
date
region
(a) Column Store with Virtual Ids
(b) Column Store with Explicit Ids
(c) Row Store

Figure 3.1: Physical layout of column-store and row-store databases. Source: [35]

MonetDB is tuned for these analytical workloads by vertically fragmenting the data, as illustrated in Figure 3.1a. This design allows operations within the database to operate on the individual columns. In a traditional RDBMS design where the data is horizontally fragmented, as illustrated in Figure 3.1c, the entire table must be scanned, even for operations that only involve a single column.

We could mimic the physical storage model in a row-store database by creating a separate table for every column. This approach has been attempted before, and while it does provide improvements over the traditional row-store database design when there are a significant amount of unused columns, it is still a lot less efficient than dedicated column-store databases [16].

The reason for this is that 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 [17]. MonetDB processes the data in a column-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 an entire column at a time. In addition, MonetDB uses many cache conscious algorithms, greatly reducing the amount of cache misses and time spent retrieving data from memory or disk.

This processing model is exactly why MonetDB is so well suited for scripting language UDFs, such as *R* and *Python*. These scripting languages rely on efficient vectorized processing. If we want to perform vectorized processing in a RDBMS that uses the typical *tuple-at-a-time* approach, we have to first iterate over all the tuples in the entire table and copy the column values into a temporary array, and then use our vectorized function on this temporary array. However, in MonetDB the vectorized function fits directly into the query execution engine.

3.1.1 The MonetDB Kernel

In this section we will describe the internal workings and several optimization techniques that MonetDB uses in more detail.

Data Storage

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 3.1a. This is accomplished by only storing the *oid* of the first element, oid_{base}, and ensuring that the subsequent values have incremental *oids*. We can then find the *oid* of the i^{th} value by computing $oid_{base} + i$. BATs with implicit *oids* are called *headless BATs*. As of recently, MonetDB only makes use of *headless BATs*.

Delta Structures

One of the key disadvantages of columnar storage is that updates to individual tuples are expensive. As the values of a single tuple are scattered across separate columns, the system must perform one I/O operation for each column it writes to.

MonetDB alleviates this issue by making use of delta structures [36]. Instead of writing the updates directly to disk, they are stored in delta structures that are kept in memory. When the columns are used in a query, the delta structures are then merged with the actual columns to ensure that the correct view of the columns is given.

The purpose of these delta structures is two-fold. In addition to making updates to the database cheaper, they provide cheap snapshot-isolation that is used for transaction management. By using delta columns, the original columns on disk can be left unmodified while still providing an accurate view of the updated columns. A separate set of delta columns is maintained for every separate transaction. When the transaction is committed, the delta columns are merged with the actual columns and written to disk.

While delta structures make updates significantly cheaper, they make subsequent operations involving these columns more costly. The base columns themselves cannot be used directly when there are updates in the delta structure. Instead, the delta structures must be merged with the base columns to create a correct view of the column.

Query Plan Execution

MonetDB translates SQL queries into a set of Monet Assembly Language (MAL) instructions. MAL instructions describe various concrete operations to perform on the BATs.

- `bat.new`: Creates a new BAT of the given type.
- `bat.append`: Append a set of values to an existing BAT.
- `algebra.subsort`: Sorts a BAT in either ascending or descending order.

The complete set of MAL instructions that execute a SQL statement is called the MAL plan. Strategic optimization takes place during this phase, such as pushing selections down and re-ordering joins based on size estimations.

Parallel Query Plans

In addition to common strategic optimizations, MonetDB has a set of optimizers for parallel query plan generation [37]. The *mitosis* optimizer partitions the largest input columns into several separate columns based on size estimation heuristics and the available amount of CPU cores and main memory. The *mergetable* optimizer then uses the partitioning scheme created by mitosis to create a parallel query plan.

When creating the parallel query plan the mergetable optimizer has to take into account the fact that certain MAL instructions cannot be executed on partial columns. Instructions such as `aggr.quantile` and `algebra.subsort` require access to the entire column to compute the correct result. These instructions are called *blocking instructions*, as opposed to *mappable instructions*.

The mergetable optimizer accounts for blocking instructions by packing together the partitioned columns before calling the blocked instruction using the `mat.pack` operation. This operation waits for all the parallelized threads to finish executing, and then packs the result columns into a single column, before passing it to the blocking operation.

Runtime Optimization

MAL instructions describe the operations to perform on the BATs, but they do not describe how to perform them. For instance, there is a MAL instruction called `algebra.leftfetchjoin`[1] that instructs the program to join two BATs together, however, it does not describe which algorithm to use to perform the join.

When the MAL instruction is executed, MonetDB decides which algorithm is more efficient based on various properties of the input BATs, such as the size and type of the BAT, whether or not it is sorted and whether or not it has an index. This process is called runtime optimization.

[1] `algebra.leftfetchjoin` will be renamed to `algebra.projection` in the next release.

3.2 Python Internals

Python is a very popular scripting language, currently ranked as the fifth most popular programming language behind Java, C, C++ and C# [38]. Python was conceived by Guido van Rossum at the CWI in the late 1980s [39]. It is either the most popular [5, 13] or second most popular [40] scripting language in data science, depending on the metric.

Python is a scripting language, meaning that the language is interpreted at run-time rather than compiled. Scripting languages are a lot more convenient for the user when used in user-defined functions. Rather than having to compile the source code and link it to the database, as the user has to do when writing a user-defined function in a compiled language, the user only has to supply the source code of the function. The source code can then be interpreted by the server at run-time.

Python's base library only provides limited functionality, but can be extended through *modules*. Modules extend the base library with new functions, classes and data types that can be used by the user. The Python standard includes an extensive set of modules that, amongst others, provide functionality for math operations, file operations, multiprocessing, http requests, encryption and parsing.

In addition to the large amount of modules included in the Python standard, there are countless modules created by users themselves. These modules do not have to be written in the Python language. Libraries that perform expensive computations are often written in *C*, and directly operate on *C* arrays wrapped in Python objects rather than operating on Python objects. This allows these modules to perform highly efficient vectorized computations on arrays.

The most frequently used modules for data science include `numpy`, `scipy`, `sympy`, `sklearn`, `pandas` and `matplotlib`. These modules offer functions for high performance data analytics, data mining, classification, machine learning and plotting.

3.2.1 Python Interpreters

To be able to understand the performance of various Python libraries and Python scripts, we must first understand the internals of the Python interpreter. While Python is often seen as interchangeable with CPython, the most popular and most used interpreter for the language, there are multiple interpreters for the language.

There are over twenty different Python interpreters [41]. However, most interpreters are not mature and not compliant to any specific version of the Python language. We will focus only on mature Python interpreters.

- *CPython* [42] is the most commonly used interpreter. It is the original Python interpreter, and is a reference implementation for most other interpreters. It is written in *C*.
- *Jython* [43] is an alternative Python interpreter implemented in Java. It compiles directly to Java bytecode, and is easy to integrate into existing Java applications.
- *IronPython* [44] is a Python interpreter written in *C#*. This interpreter is easy to embed into *C#* applications.

- *PyPy* [45] is a Python interpreter written in *Python*. It is designed to be a very fast interpreter, that supports multithreading and JIT compilation.

For this implementation, we have chosen *CPython* for two different reasons. A large amount of packages for Python are written as extensions to *CPython*, such as the very popular *NumPy* and *SciPy* packages. These are written in *C* for performance reasons and use the *CPython C API* [46]. These packages are not compatible with other Python interpreters. In addition, as we are working with MonetDB, integration with *CPython* is more efficient, as they are both written in *C*.

3.2.2 The CPython Kernel

We will now briefly examine the *CPython Kernel* as the internals of the interpreter are important when considering the performance of Python code.

Every Python object in *CPython* is an extension of the base class `PyObject`, including basic types such as integers, strings and floats.

```c
typedef struct {
    ssize_t reference_count;
    struct typeobject *type;
} PyObject;
```

The base object `PyObject`, and thus every Python object, contains two variables. The reference count, and a reference to the type object.

The reference count is a counter used for garbage collection purposes. When an object depends on a specific Python object, it increases the reference count, and when it no longer needs the object it decreases it. When the reference count reaches zero, the object is deleted.

The reference to the type object determines the type of the Python object. The actual typeobject contains all the relevant information about the type, including the constructors and destructors, the individual attributes, methods of the object and information on how the object interacts with various operators.

Aside from the base information, Python objects can contain additional information. As an example, we will examine the *integer* type in Python, called the `PyIntObject` in *C*. Note that the *integer* type is represented by a 64-bit long integer.

```c
typedef struct {
    PyObject_HEAD
    long value;
} PyIntObject;
```

The actual integer value is stored in the *value* variable. Since *C* does not support object inheritance, every Python object manually includes the header information by including `PyObject_HEAD`.

There are two major performance implications of this system. The first is that all Python objects carry the inherent overhead of the `PyObject` structure. On a 64-bit system, this overhead is 16 bytes, as pointers are 8 bytes. If we want to represent a single 64-bit integer in Python, we need 24 bytes of memory, three times more than the 8 bytes necessary for storing the actual integer value.

The second is that since all objects are individual Python objects with an individual reference count, it has to be possible to delete every Python object individually on the heap. This means that we have to allocate Python objects individually, as the Python object has to be responsible for its own chunk of memory.

Thus we need to perform a call to `malloc` for every Python object we create. The result of this is that creating a large amount of Python objects is very inefficient, as we have to allocate a large number of small objects on the memory heap.

3.2.3 Global Interpreter Lock

The CPython interpreter was built with the Global Interpreter Lock (GIL). The interpreter does not perform fine-grained locking of Python objects. As an example, incrementing and decrementing the reference count of an object are done through simple addition and subtraction, rather than by using atomic operations. The interpreter was built with the Global Interpreter Lock because fine-grained locking is slower in single-threaded applications, and single-threaded execution allows for easier integration of existing *C* libraries that might not be thread safe [47].

The GIL significantly limits the possibilities of multithreading in Python. Interpreted Python code cannot be run in parallel within the same process, as any thread running Python code must hold the GIL. Note that a limited amount of parallelization is still possible, as the GIL is released when performing I/O operations or when performing large computations in *C* code, allowing other threads to run interpreted Python code.

Chapter 4

Implementation

In this section, we will give an in-depth description of our system, MonetDB/Python. We will explain how users can create and use MonetDB/Python functions. We will describe the implementation of our system, the architectural decisions we have made while implementing the system and how they influence the performance of MonetDB/Python functions.

4.1 Usage

MonetDB/Python is built on MonetDB, a column-store database. MonetDB uses *operator-at-a-time* processing. As such, MonetDB/Python differs from user-defined functions in *tuple-at-a-time* databases.

In *tuple-at-a-time* databases, user-defined functions are either called once per row, as in MySQL or Postgres, or the user-defined function requires the user to iterate over the rows manually, as in AsterData or Vertica. In both cases, the user-defined function must process the rows individually. This is the case because the database processes data one tuple at a time.

In *operator-at-a-time* databases, the operators process the entire columns at once. MonetDB/Python behaves in the same way. As such, sequential MonetDB/Python functions are called once with the entire columns as input.

MonetDB/Python supports parallelized execution of user-defined functions. When the user specifies that the MonetDB/Python function can be parallelized, the input columns will be partitioned into several pieces, and the function will be executed once for each partition.

Query Syntax

In this section, we present the syntax for creating and using the various types of MonetDB/Python functions.

User-Defined Scalar Functions

Scalar functions can be used almost anywhere built-in SQL functions can be used. The syntax to create a user-defined scalar function is shown in Listing 4.1.

```sql
CREATE FUNCTION fname([paramlist] || *)
RETURNS <return_type>
LANGUAGE PYTHON || PYTHON_MAP
{ function_code } || 'external_file.py'
```

Listing 4.1: Syntax for creating user-defined scalar functions.

Scalar functions can either take a fixed number of input columns as input, specified when creating the function, or take an arbitrary number of input columns. To create a scalar function with a fixed set of input columns, the input columns must be specified as a set of (name, type) pairs when the function is created. To specify that the function is polymorphic and can take any number of arguments, the input list should be replaced by an asterisk.

Scalar functions always return a single column. The type of the column that the scalar function returns must be specified when the function is created.

Scalar functions can either be run sequentially, or in parallel. The user has to specify this when creating the function by setting the language of the function to either `PYTHON`, indicating the function has to be run sequentially, or `PYTHON_MAP`, indicating the function can be run in parallel.

Finally, the user must specify either the source code of the function, or provide the location of a file to load the source code from. The source code is loaded from the specified file every time the function is executed, not when the function is created, allowing the user to alter the source code of the function by modifying the file without having to restart the database or recreate the function.

```sql
SELECT func2(func1(i, j), k) FROM some_tbl;
```

Listing 4.2: Nested scalar functions.

Scalar functions can be nested, either with built-in SQL functions or with other MonetDB/Python functions. An example of how scalar functions can be nested is given in Listing 4.2.

User-Defined Aggregate Functions

Aggregate functions can be used anywhere built-in SQL aggregates can be used. The syntax to create a user-defined aggregate function is shown in Listing 4.3.

```sql
CREATE AGGREGATE fname([paramlist] || *)
RETURNS <return_type>
LANGUAGE PYTHON || PYTHON_MAP
{ function_code } || 'external_file.py'
```

Listing 4.3: Syntax for creating user-defined aggregate functions.

The difference between aggregate and scalar functions is smaller than in typical *tuple-at-a-time* databases, as users are given access to the entire input columns in both cases. The main difference between the two is that aggregations can be used in combination with the `GROUP BY` clause, to perform aggregations over multiple groups.

When the aggregation uses the language `PYTHON_MAP`, the aggregation is computed in parallel over the set of groups, such that the user-defined function is called once for every group with the rows of the individual group as input. The execution of the set of user-defined functions is split over n threads, where n is either specified by the user or equal to the amount of cores in the machine.

When the aggregate is set to use the language `PYTHON`, the aggregation function is only executed once. To be able to aggregate over the different groups, an additional input column called `aggr_group` is passed to the aggregate. This column specifies, for each row, to which group that row belongs. The user-defined function must then return a single aggregated value for every group.

User-Defined Table Functions

Table functions are functions that return a set of columns (i.e. a table), rather than just a single column. As such, they can be used at any place where regular tables can be used. The syntax to create a user-defined table function is shown in Listing 4.4.

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

Listing 4.4: Syntax for creating user-defined table functions.

The main difference between scalar functions and table-producing functions is that table-producing functions can output multiple columns, rather than just a single column. Table-producing functions can be parallelized.

```
SELECT * FROM func( (SELECT * FROM some_tbl) );
```

Listing 4.5: Query input to table function.

The input to a table-producing function can be either nothing, a set of scalar values, or the result of a query as a set of columns, as shown in Listing 4.5

4.2 Implementation

In this section, we will describe the architectural decisions we have made while implementing MonetDB/Python. For each architectural decision we have made, we have examined the possible alternatives and have run microbenchmarks to support our choices.

MonetDB works with MAL instructions internally, we can extend MonetDB to include Python support by introducing a new MAL function `pyapi.eval`. Like other vectorized MAL functions, this function takes a set of BATs as input (the input columns), and outputs a set of BATs (the result columns).

The general pipeline of the MAL function is to take the input BATs and convert them into a set of Python objects that are usable within the Python function. We then compile and execute the Python UDF with the converted input as the set of parameters. Finally, we take the return values of the Python function and convert them back into a set of BATs.

SQL Type	Storage	Python Object	Object Size
BOOL	1 Bytes	PyBoolObject	24 Bytes
SHORT	2 Bytes	PyIntObject	24 Bytes
INTEGER	4 Bytes	PyIntObject	24 Bytes
LONG	8 Bytes	PyIntObject	24 Bytes
FLOAT	4 Bytes	PyFloatObject	24 Bytes
DOUBLE	8 Bytes	PyFloatObject	24 Bytes
STRING	n Bytes	PyStringObject	40+n Bytes
STRING	n Bytes	PyByteArrayObject	48 Bytes

Table 4.1: The memory overhead for various SQL types.

4.2.1 Input Conversion

When our function is called, it takes a number of input columns, these are internally represented as BATs. The first step in our pipeline is to convert these input BATs to a set of Python objects that can be accessed by the user within the Python function.

Naive Solution

The naive way of making the BATs accessible in Python code would be to create a `List` filled with Python objects. For a column that holds 32-bit integers, we would create a `List` of `PyIntObjects`.

This is extremely inefficient, however. Since CPython does not allow us to allocate a batch of Python objects, we need to allocate them individually. Which means we must perform a `malloc` call for every single row in our column.

This method also requires significantly more memory. Each Python object has additional overhead associated with it, as the object has to keep track of its type and its reference count (for garbage collection purposes).

Even in the best case, our memory usage increases by a factor of three. When dealing with large columns, we would be using a significant amount of extra memory.

Column Objects

We have argued that creating a single Python object for each row is very undesirable. Not only is this conversion very inefficient, it also significantly increases memory usage of our application. However, we still need to make our columns accessible in Python. We do not need to make a Python object for each row in each column. Instead, we could create a Python object for each individual column. This way the overhead of creating Python objects does not scale with the size of our dataset but only with the amount of columns.

We can define a Python Object that holds the values of an integer column.

```c
typedef struct {
    PyObject_HEAD
    int *column_values;
} ColumnObject;
```

Since we now only have a single Python object for each column, the overhead of creating Python objects is now significantly lower.

However, with this simple structure we cannot actually use the integers stored in the column within Python yet. We could make them available to us in Python by overloading the array access operator. We can then access *i′th* integer by calling `ColumnObject[i]`.

But now we encounter the same problem as we did previously. If the user calls `ColumnObject[i]` in Python, we must return a Python object. Thus we must convert the *i′th* integer into a `PyIntObject`. So if the user accesses the entire column, we are still converting every single value into a separate Python object. Worse still, if the user accesses the same element multiple times, we have to convert the same element multiple times.

NumPy Arrays

The only way in which we can avoid creating Python objects for every element is if we never transfer the individual elements to Python at all. We could accomplish this by creating native functions in *C* that take as input a `ColumnObject` and operate on the underlying values in some way. This is exactly what the NumPy [48] and SciPy [49] libraries do. At the heart of these libraries lies the NumPy Array.

```c
typedef struct {
    PyObject_HEAD
    char *data;
    int nd;
    int *dimensions;
    int *strides;
    PyArray_Descr *type;
    int flags;
} NumpyArray;
```

The NumPy array is a more advanced version of our `ColumnObject`, which includes information about the dimensions of the array, the type of data that is stored in the array and various flags that specify various properties of the data. Just like the `ColumnObject`, we only need to create a single NumPy array for each column as it holds all the data of the column.

Note that NumPy arrays still suffer from the same problem as the `ColumnObject`. If we want to access the individual elements in Python, we still have to convert the internal elements to Python objects. However, since NumPy arrays have extensive support for mathematical operations and there are numerous libraries that operate directly on the internal data for complex tasks, such as `sklearn` [50], `pandas` [51] and `sympy` [52], this is rarely required.

BAT to NumPy Array

The question is then how can we efficiently construct a NumPy array from a BAT. The easiest solution would be to create a NumPy array and copy over all the data, however, this means we need to do a complete scan of every column and this also means that the same column now exists twice in memory. Once in the BAT, and once in the NumPy array.

Since both NumPy and MonetDB are written in *C* and internally use the same representation of numeric values, we do not need to copy the values to NumPy. Instead, we can create a NumPy array that directly references the data inside the BAT.

This carries a risk with it, though. As we are giving the NumPy array direct access to the internal data of the BAT, we could possibly modify the database through this NumPy array. Consider the following simple SQL query.

```sql
SELECT python_function(i) FROM integers;
```

If `python_function` modifies the input array, the function would modify the data in the actual database. This could easily be done by accident when the user calls functions that modify the contents of the array, such as the `numpy.random.shuffle` function that shuffles an array in-place. This is undesirable as it would cause unpredictable behavior.

Fortunately, NumPy arrays have a read-only flag that prevents modification of the data in Python. The function `numpy.random.shuffle` will throw an exception when used with a read-only array, and the user would first have to create a copy of the array to use that function.

This only partially solves the issue. While the NumPy and SciPy modules respect this read-only flag, the NumPy array still holds a pointer to the internal database data in *C*, and *C* is not a safe language. A Python module written in *C* could choose to ignore this flag, either on purpose or by accident.

We could create a copy of the columns to be on the safe side, however, this costs a significant amount of time compared to not copying them. In addition, we would use additional memory to hold a copy of a column that already exists in memory.

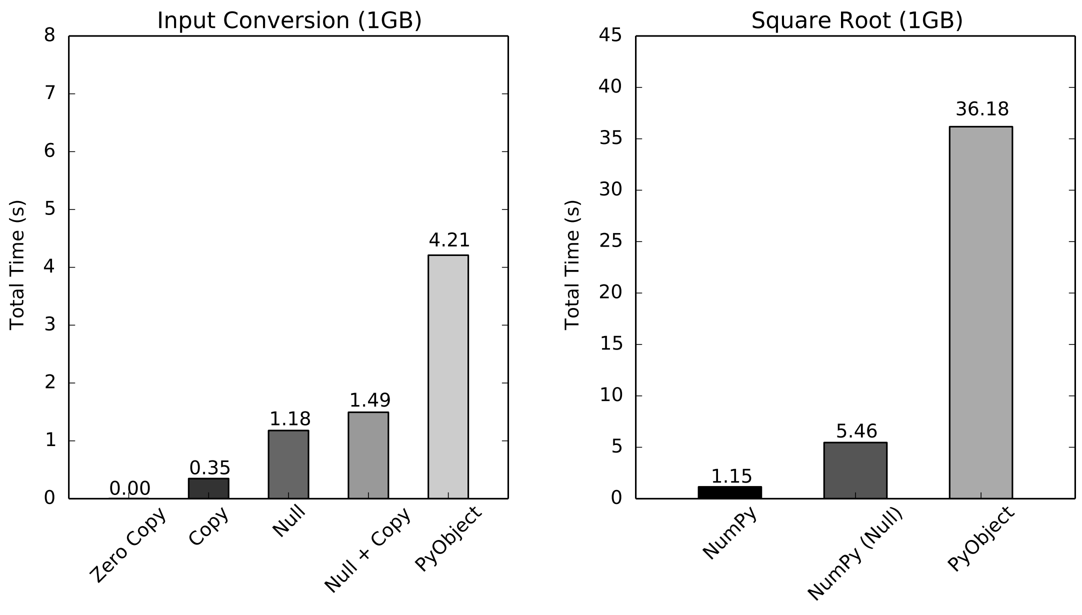
Since any user-created Python modules must be installed on the server, malicious Python modules can only be installed by people that have internal access to the server already, in which case copying the columns would not prevent the server from being in danger. And since the official modules in the NumPy and SciPy stack respect the read-only flag, the risk taken by not copying the columns is very low.

Input Conversion Benchmark

To determine the cost of copying the input data, we have run a benchmark and measured the total conversion time on a column holding `1GB` of integers. We measured the difference between the *zero-copy* and the *copy* approach. In addition, we have tested how creating Python objects stacks up against creating a NumPy array from the data.

We expect the *zero-copy* approach to take very little time, as we are only reassigning a pointer, which is an $O(1)$ operation. The *copy* operation should take significantly longer, as we have to scan the entire array, which is an $O(n)$ operation. While converting to a set of `PyIntObjects` is also an $O(n)$ operation, there is significantly more overhead involved as we have to perform a `malloc` call for each input row.

The results of the benchmark are shown in Figure 4.1a. As we can see, the *zero-copy* approach takes a negligible amount of time. Surprisingly, the *copy* approach is more expensive. However, the difference is not as big as we might have expected. This is because we copy the data using the `memcpy` operation, which is highly efficient. We can see creating `PyObjects` is significantly slower than either operation. It is over ten times as slow as copying the input array using `memcpy`. This is because manually looping over the input array and calling `malloc` for each row is very expensive.



(a) Time taken to convert the data. (b) Time taken to compute square root.

Figure 4.1: Benchmark: Different conversion methods of 1GB of integers.

The difference between NumPy arrays and Python objects extends further than just the conversion costs. The performance of operations involving them is also affected because the data within NumPy arrays is packed tightly in memory, whereas the data within Python objects is scattered throughout memory. To test the performance difference between computations involving the different representations, we have computed the square root of a column holding `1GB` of integers using the different representations. Note that we use the vectorized NumPy operation `numpy.sqrt` on the NumPy array, and the Python operation `math.sqrt` on the set of Python objects.

As the NumPy operation internally loops over a *C* array, we expect it to be highly efficient. We expect `math.sqrt` to be significantly less efficient, as it operates on a set of Python objects, meaning it has to iterate over a significantly larger chunk of memory. Unlike the *C* array, the Python objects are not packed tightly into memory, further increasing the cost of scanning through the entire array.

As we can see in Figure 4.1b, computing the square root with the non-vectorized operation is extremely slow. Not only does it operate on the Python objects scattered through memory, but we are calling a Python function once for every element, and Python functions have a much higher overhead than *C* functions. As a result, the `numpy.sqrt` operation takes only a fraction of the time that `math.sqrt` takes.

Handling Missing Data

While NumPy and MonetDB both use *C* arrays to store numeric values, they have a different representation of missing data. MonetDB does not have a special flag for *null* values, but stores them directly in the column. The way in which missing data is encoded depends on the type of the column. The convention is that the lowest possible value is interpreted as *null*. In a column holding 32-bit integers, the value -2^{31} represents the *null* value.

NumPy uses a different representation. Instead of directly encoding the *null* values in the column, they use a separate Boolean array that indicates, for each individual value, whether or not that value is null or not. This separate array is called a *mask*.

Thus if we use the above approach to create a NumPy array from a BAT, the *null* values will not be correctly converted. For NumPy to properly deal with *null* values, we have to explicitly construct the mask. The only way in which we can do this is by scanning the input BAT and checking, for each value, if it is equal to *null*, and then storing that in the mask.

Computing this mask will take a significant amount of time, and degrade the performance of our input conversion. Instead of our zero copy $O(1)$ operation, we have to perform an $O(n)$ scan over the input array.

Operations on a masked array are slower as well, especially since support for *null* values in NumPy is not deeply integrated into the library. Rather than checking during computation if a value is *null* and then skipping the computation, the computation is made for every value in the array, and then the *mask* is reapplied. As a result, operations on arrays involving *null* values in NumPy are less efficient than they could be.

As we wanted to test how much the presence of *null* values influences the performance of both the conversion and computations, we ran the same benchmarks that we did for the `PyObject` and regular NumPy array. We measured how long it took to convert `1GB` of integers to a NumPy masked array, and we measured how much time performing the `numpy.sqrt` operation on the same `1GB` of integers took. We expect the NumPy masked array to be significantly slower than the regular NumPy array, in both the conversion time and the `numpy.sqrt` operation.

As we can see in Figure 4.1a, converting an array with *null* values is significantly more expensive than copying the data, as we have to loop over the data rather than calling `memcpy`, and we have to perform a check for every value. We can also see that operations on masked arrays are significantly more expensive, as the `numpy.sqrt` operation takes significantly longer to complete.

We could choose to always create a NumPy masked array for consistency. However, as both conversion to and operations on masked arrays are significantly less efficient, we would prefer to only construct the *null* mask when the column we are converting actually has *null* values. Fortunately, every BAT has a flag that indicates whether or not it contains any *null* values. If the flag indicates that the BAT does not contain *null* values, we do not have to do any unnecessary work.

Strings

In the previous section, we have shown how we can create a NumPy array from a BAT without copying any data. However, this approach only works with numeric values, because both MonetDB and NumPy use regular *C* arrays to represent a sequence of numeric values. As strings are arrays of characters, an array of strings is essentially an array of arrays, and as such they are significantly more complex to store. MonetDB and NumPy use a different storage method for strings.

MonetDB String Heap

MonetDB stores strings using a separate string heap. The string heap is a heap-like structure that contains the actual string values, while the main BAT contains references to the strings of each element. The advantage of this storage solution is two-fold. As the BAT contains references to the strings, duplicate strings can be eliminated. Instead, two rows that contain a reference to a duplicate string can simply point to the same position in the string heap. This significantly reduces storage space when there are many duplicates. In addition to duplicate elimination, the string heap allows MonetDB to efficiently store strings of different sizes without specifying a fixed maximum size. This allows MonetDB to support the SQL types `STRING` and `BLOB`, as opposed to only supporting the SQL type `VARCHAR (n)`.

NumPy Strings

A NumPy string array stores the strings consecutively in a multidimensional array by assigning a fixed maximum length m for all strings in the array. When first creating the array of strings, this maximum length must be specified. Every string then occupies m bytes of storage, regardless of how long the string actually is. The total memory usage of the NumPy string representation is shown in Equation 4.1, where n is the amount of strings and l_i is the length of string i in the array.

This representation works optimally when all the strings have the same size, as there is zero wasted space and the strings are all tightly packed together in memory. However, when there is one very large string of length m and a large number of small strings, the memory usage of this string representation explodes, as every small string still requires m bytes of memory.

$$n * \max_{0 \leq i \leq n}(l_i) \tag{4.1}$$

String Objects

The alternative to using the NumPy string representation would be to create a batch of Python Objects to represent the individual strings. There are two separate Python Objects that can represent strings. The standard object is called the `PyStringObject`. This object holds the actual string and meta information about the string. The memory overhead of the metadata in the `PyStringObject` is 40 bytes, as shown in Table 4.1. The actual string requires l bytes of storage for a string of length l. The total memory usage of this representation is shown in Equation 4.2.

$$\sum_{i=1}^{n}(40 + l_i) \tag{4.2}$$

Byte Array

There is another object that can represent strings in Python. This object is called the `PyByteArrayObject`. The difference between the `PyByteArrayObject` and the `PyStringObject` is that the `PyByteArrayObject` holds a reference to a string, instead of directly storing the string. As we already have the strings stored in the string heap of MonetDB, the `PyByteArrayObject` allows us to make the string accessible in Python without storing the same string twice. This can save a significant amount of storage space when dealing with large strings, and has the added benefit of not requiring a significant amount of extra memory when storing duplicates of large strings. However, the `PyByteArrayObject` is less efficient when dealing with small strings, as a pointer requires 8 bytes of storage (on 64-bit systems), while short strings can be stored in less than 8 bytes using the `PyStringObject`. The total memory usage of this representation is shown in Equation 4.3.

$$n * 48 \tag{4.3}$$

Unicode Objects

All of the mentioned string objects can only hold ASCII strings. These string formats are not suitable when there are Unicode characters in the database. To support Unicode characters, Python has the `PyUnicodeObject`. The `PyUnicodeObject` functions similarly to the `PyStringObject`, but instead of holding single byte characters encoded in ASCII, `PyUnicodeObjects` hold four-byte Unicode characters encoded with UCS4. Thus they require significantly more memory, as illustrated in Equation 4.4.

$$\sum_{i=1}^{n}(40 + 4l_i) \tag{4.4}$$

Supporting Unicode strings adds another challenge when converting between MonetDB and Python types, as MonetDB uses UTF8 encoding, and Python uses UCS4 encoding. When using the `PyUnicodeObject`, not only do we have to copy the strings, we have to convert between the different encodings used.

When the input column has no Unicode characters, we can avoid doing this work and construct a `PyStringObject` instead. However, we must first determine if the input has no Unicode characters. This means before we can start converting to a string, we must first scan the input to check if there are Unicode characters present. Alternatively, we can neglect this step and always create a `PyUnicodeObject`.

Benchmarks

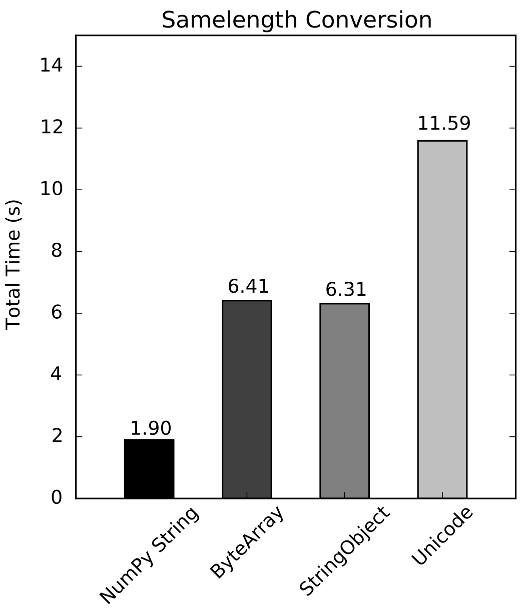
We have illustrated the different possible storage methods for strings in Python, and their advantages and disadvantages. There is no clear best solution, so we will measure how each of the solutions perform in various circumstances.

Note that all the benchmarks are performed on randomly generated alphanumeric strings, as we must use the `PyUnicodeObject` when there are unicode characters in the database. When using the other representations, we first scan the string column and check if there are any Unicode characters. After confirming that there aren't any, we use the `PyStringObject`, `PyByteArrayObject` or NumPy string.

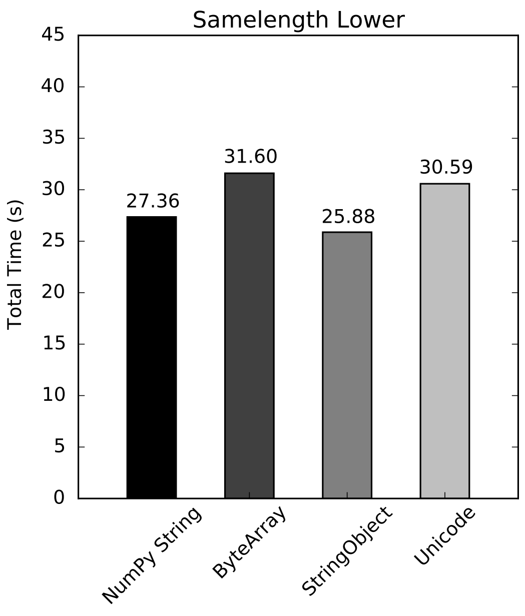
When using the `PyUnicodeObject` in the benchmarks, we skip the initial scan and immediately construct a set of `PyUnicodeObjects`.

Same Length Strings

The first benchmark we will perform is measuring how each of the different storage methods perform when dealing with a set of strings that each have the same length. This is the best case scenario for the NumPy string representation, as there is no wasted space. In the benchmark, we used a column with `1GB` of strings, each with a length of 10 characters. The strings were randomly generated, and could only contain alphanumeric characters. In Figure 4.2a, the conversion time of the different types of string arrays are shown.



(a) Convert a column of strings.



(b) Convert column of strings to lower case.

Figure 4.2: Benchmark: A column of strings with the same length.

Conversion times are not the only consideration when choosing a string representation. One of the main advantages of the NumPy string representation is that the strings are directly adjacent in memory, whereas the Python Objects are scattered in memory as they are separately allocated. As a result, we expect that string operations on the NumPy array are more efficient as well. We test this hypothesis by performing the `lower` operation on every string in the array. This operation performs a single scan through the entire string array and changes every uppercase character into the corresponding lowercase character. In Figure 4.2b, the execution time of the `lower` operation on an array of same length strings is shown.

As we can see in Figure 4.2a, the NumPy string representation indeed has the fastest conversion time. However, the difference is a lot smaller than the difference between the NumPy array and Python objects for numeric values. The reason for this is that we cannot do a single `memcpy` to copy the strings, but we have to copy the strings from the MonetDB string heap one by one. This is a lot more expensive than performing `memcpy` on a single array.

We can see that the `PyUnicodeObject` object takes significantly longer to construct than all the other string representations. This is because we have to convert every single character from UTF8 to UCS4 representation, whereas we only need to copy the characters when using the other representations.

As we can see in Figure 4.2b, the difference between the different representations when performing computations is low. This is surprising, as we expected the NumPy string representation to be much more efficient when performing computations. Looking at the NumPy source code reveals the reason for this performance difference. While NumPy offers a set of string operations, they are not actually implemented in C, but rather convert the array to a set of Python objects and then call the Python operation. As such, when we actually perform the `lower` operation, it is slower than the `PyStringObject` representation. When using the NumPy representation we first convert to the NumPy representation, and then to the `PyStringObject` representation, whereas when using the `PyStringObject` we immediately convert to that representation.

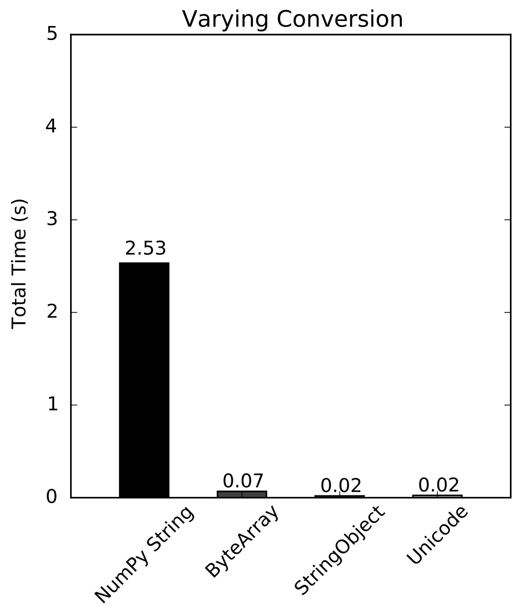
We also observe a performance difference between the `PyStringObject` and the `PyByteArrayObject`. This is because the `PyByteArrayObject` has two levels of indirection. The array references the `PyByteArrayObject`, and the `PyByteArrayObject` references the string in the string heap. The `PyStringObject` directly holds the actual string. As a result, the `PyStringObject` is more efficient when the data is actually being used, especially for a large amount of smaller strings, as is the case here.

Varying String Lengths

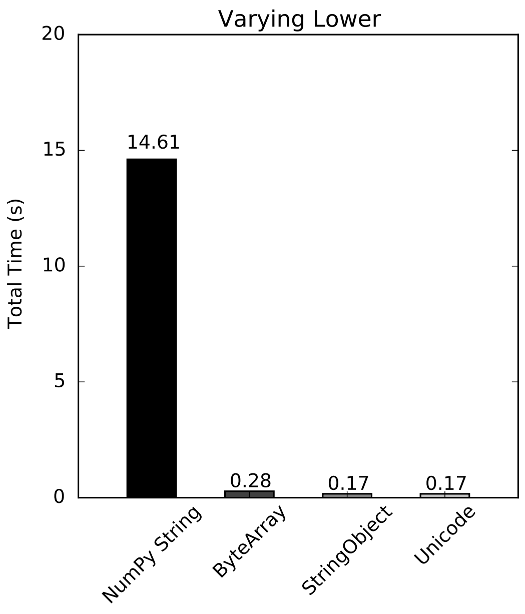
The previous benchmark was the best case scenario for the NumPy string representation. The NumPy string representation does not deal well with data sets with varying string lengths. Especially when there is a small amount of extremely long strings, and a large amount of short strings. In this benchmark, we test how well each string representation performs when exactly this scenario occurs. In this scenario, the database holds one large string of a fixed length of $10,000$ characters in a table with $1,000,000$ entries. Every other entry contains a string of length 1. In Figure 4.3a, the conversion time for the different types of string arrays is shown.

As in the previous benchmark, we will perform the *lower* operation on every string in the array. With this dataset, the NumPy string representation has a lot of unused space and is not very densely packed, thus we expect it to perform significantly worse than in the previous benchmark. We don't expect the performance of the *lower* operation with the Python Object representation to change significantly compared to the previous data set, as the varying string lengths should not influence how tightly packed the data is in memory.

As we can see in the benchmark results in Figure 4.3a, converting the data to the NumPy string representation takes significantly longer. Converting to the other string representations is much faster, as the total size of the strings is only about `1MB`. However, as there is a single string that has a size of `10KB`, and there are $1,000,000$ strings, the NumPy string representation uses `10GB` of memory to store



(a) Convert a column of strings.



(b) Convert column of strings to lower case.

Figure 4.3: Benchmark: A column of strings with varying length.

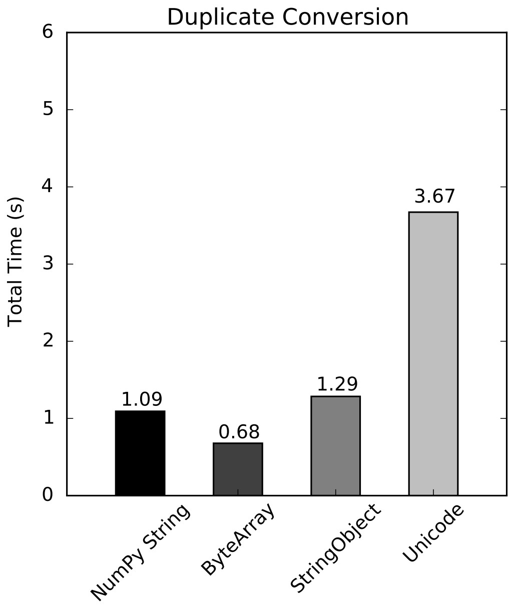
the `1MB` of strings. Allocating this large amount of memory and copying the string values into it takes a significant amount of time. Even though the individual strings are small, they are spread across the entire range of the `10GB` of memory, requiring us to scan through the entire `10GB` to convert to this representation.

The performance difference is even more pronounced when we actually use the NumPy array and perform the `lower` operation. The operation completes within a fraction of a second when using the other string representations, however, it takes almost 15 seconds when using the NumPy array representation.

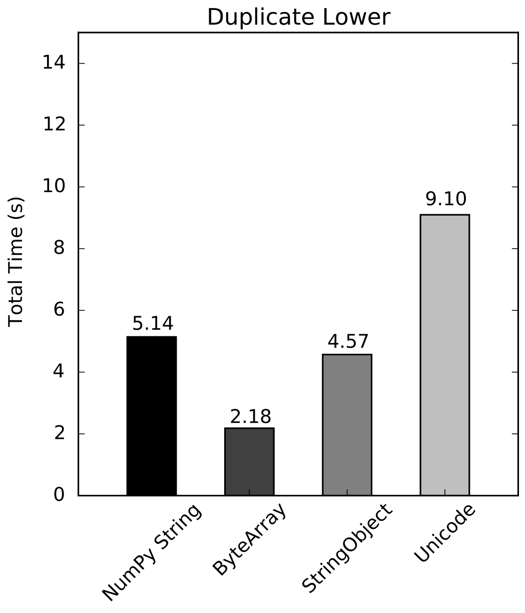
Duplicate Strings

The memory usage of both the NumPy string representation, the `PyStringObject` and the `PyUnicodeObject` representation scale with both the length of the individual strings and the total number of strings. This can lead to very high memory usage when dealing with large strings. It is especially bad when dealing with a large string that has been duplicated, as this can lead to memory usage exploding unexpectedly. Consider a single `1MB` string that has been duplicated over 1000 rows. As the MonetDB string heap performs partial duplicate elimination, the string heap will only contain the string once and occupy `1MB` of memory. If we convert this string column using the NumPy string representation or the `PyStringObject` representation, we will materialize the duplicated string once for every row, leading to `1GB` of memory usage. This is precisely where the `PyByteArrayObject` representation shines, as it will not materialize the duplicates, but will only create a reference to the string heap.

In this benchmark, we repeat the same operations as the previous two benchmarks. However, this time we use a column that holds 1000 duplicate strings of `1MB` (or $1,000,000$ characters) each. As the NumPy representation and the `PyStringObject` representation copy the individual strings, we expect them to take significantly longer to construct than the `PyByteArrayObject`. And as both use significantly more memory than the `PyByteArrayObject` representation, we expect operations on those representations to be significantly slower as well, as we will have to scan a much larger chunk of memory to perform the operation on all the strings.



(a) Convert a column of strings.



(b) Convert column of strings to lower case.

Figure 4.4: Benchmark: A column holding a single string duplicated over all rows.

As we can see in Figure 4.4a, the `PyByteArrayObject` has the fastest conversion time. The time taken in the conversion of the `PyByteArrayObject` is almost entirely spent scanning to check for Unicode characters, as the actual creation of 1000 objects is very small. The NumPy string and `PyStringObject` are not that far off. The `lower` operation is also significantly faster on the `PyByteArrayObject` representation. This is because we are repeatedly scanning the same `10MB` string, which means it can be kept in the cache. Whereas we have to scan multiple different copies of the string when performing the operation on the other representations.

Implementation and Implications

Theoretically, we could extend NumPy arrays to directly support the MonetDB string heap, which would solve the data transfer issue. However, not only would we have to ship our custom version of NumPy and be responsible for keeping it up to date, our modified NumPy array would not be compatible with a large part of the NumPy and SciPy stack of functions, as these directly use the internal *C* array for efficiency. This means that various modules and libraries will not work with our custom NumPy array, which is not acceptable as the main reason data scientists use Python for analysis is the large amount of modules and libraries available for the language. Thus we will need to convert the MonetDB string heap to Python.

The initial decision we should make is if it is worth it to scan the input array for any Unicode values, or if it is better to always create `PyUnicodeObjects`. As we can see in Figure 4.2a, converting to `PyUnicodeObject` takes significantly longer, even though we can skip the initial Unicode character scan.

In addition, as seen in Figure 4.4b, operations on `PyUnicodeObjects` take significantly longer as well. Thus the scan does not only reduce conversion costs, it speeds up subsequent operations as well. As such, performing the scan and selecting an ASCII string representation when there are no Unicode characters is the best choice.

Then we still need to choose between the different ASCII string representations. As shown in Figure 4.2a, converting to the NumPy string representation is more efficient when used with strings of similar length. As a single column in a database holds similar information, this is a common scenario. Similar data often has a similar length. The classic example in which the NumPy string representation is efficient would be a column storing gender as either M or F. However, actual operations on the NumPy array are slower, as seen in Figure 4.2b.

In addition, there could be real-life scenarios in which the NumPy string representation would be highly inefficient. There could be a column containing binary data of images, that contains a *null* value on most rows, and encoded image data on a small amount of rows. In this scenario, the NumPy string representation would fail. If there is a single image in the database that takes 1MB of storage in a table with 1000 rows, we would have to use `1GB` of memory to store a single `1MB` image. It is not a robust representation. Even in the optimal case, the benefits of using the NumPy array are small as computations involving the NumPy array are slower, rather than faster.

The `PyStringObject` representation is faster than the `PyByteArrayObject` representation when dealing with many small strings. In addition, the `PyStringObject` is the default string type used in Python. As a result, there are a lot of external libraries that do not support the `PyByteArrayObject`, and only work with `PyStringObjects`. For these reasons, the `PyStringObject` is the best choice and the one we adopted.

4.2.2 Output Conversion

Just as we had to convert the input columns to make them accessible to the user in Python, we have to convert the output columns to make them accessible to the database. This introduces a new set of challenges, as we are not in control of what type of Python Object the user-defined function returns.

The user-defined function can return either a single column, or a set of multiple columns (when it is a table-producing function). The exact output types of the operator are specified by the user when the function is created, and are fixed. This means that we must try to convert the returned objects to the correct output types, as specified when the function was created. If this is not possible, we need to raise an exception.

The most efficient output conversion occurs when the function returns a numeric NumPy array that matches the correct output type, e.g. when the expected output is the SQL type `integer` and the function returns a NumPy array of type `int32`. In this special case, the underlying C array types match, and we can apply the same trick that we perform during input conversion. We can take the array from the

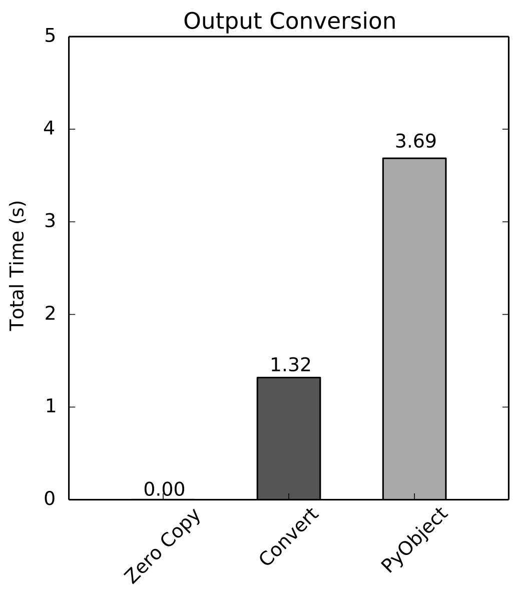


Figure 4.5: Benchmark: Output conversion of 1GB of integers.

NumPy object and use it directly in the resulting BAT. In this case, the conversion time from NumPy array to BAT takes $O(1)$ time.

Note that we can only take the data from the NumPy array if the NumPy array owns the data. If the NumPy array does not own the data, which occurs when the NumPy array is one of the input columns, we make a copy instead.

If the user returns a numeric NumPy array that does not match the correct output type, e.g. when the expected output is the SQL type `integer` and the function returns a NumPy array of type `float32`, we have to convert the output array to the expected type. This operation is an $O(n)$ scan that creates a new array and casts the individual values to the correct output type. In this case, we also print a warning to the user so they are aware of the performance loss.

The user can also return a set of Python Objects instead of returning a NumPy array. In this case, we also have to do an $O(n)$ conversion. However, as the Python objects are not tightly packed in memory and have additional memory overhead this operation is more costly than converting from a NumPy array.

To measure the difference between these output types, we measured how long the output conversion of `1GB` of integers took. In each function, we generated `1GB` of data using the `numpy.zeros` function, however, we altered the `type` of the NumPy array to either `int32`, `float32` or `object` to represent each of the different scenarios.

The function expects a 32-bit integer as the return-type, so a NumPy array of type `int32` will not require any data conversion. By contrast, the `float32` array must be converted to a 32-bit integer array. The `object` array requires us to iterate over the individual Python objects to convert the array to a 32-bit integer array.

The benchmark in Figure 4.5 shows the time taken to perform output conversion for each of these methods. As expected, the conversion of the `int32` array completes instantly. The conversion from the `float32` array to a 32-bit integer array takes a significant amount of time, and the conversion of the `object` array takes even longer.

Unfortunately, we cannot change the data type returned by the user. However, when the user returns a data type with high conversion overhead, we warn the user indicating that the performance of the user-defined function might be improved if a type with less conversion overhead is returned.

4.2.3 Parallel Execution of UDFs

Data scientists often work with large data sets. Analyzing that data can take a significant amount of time if it is done on a single core. Meanwhile, modern CPUs are scaling horizontally by adding more and more cores, as scaling vertically by improving single cores becomes more and more costly. This is why most modern databases utilize multiple cores to execute their queries. There are two different types of parallelization in databases [53].

Interquery Parallelism means that separate queries from different clients can execute in parallel.

Intraquery Parallelism means that the database utilizes multiple cores to execute a single query.

MonetDB provides interquery parallelism through the use of fine-grained locking on tables and snapshot isolation, this allows multiple clients to concurrently read data from the database.

MonetDB provides intraquery parallelism through the *mitosis* and *mergetable* optimizers. These optimizers physically split up any columns that are used in the query and processes the separated parts of the column in parallel. However, certain operations cannot be parallelized in this manner because they require access to the entire column. These are called blocking operations. An example of such a blocking operation is the quantile function, because we cannot compute the quantile of a column when we only have access to part of a column.

By default, MonetDB/Python functions are blocking operations, because it is very easy to write a user-defined function that is not parallelizable. We could, for example, use the user-defined function to compute a quantile. The user can manually specify when creating the user-defined function if the function can be safely parallelized, or if it should be executed as a blocking operation. If it is not a blocking operation, the input columns will be split up and the user-defined function will be executed in parallel.

However, there is an obstacle to running MonetDB/Python functions in parallel. As mentioned in Chapter 3.2, the CPython implementation suffers from the *Global Interpreter Lock* (GIL).

Any thread that runs Python code must hold the GIL. As a result, interpreted Python code can only be run in a single thread for each process. Note that the GIL is released when performing I/O operations and when performing large computations in libraries implemented in *C*, such as NumPy. This means that if a MonetDB/Python function uses functions that release the GIL, the GIL will not prevent all parallel execution of the function. However, there will always be some interpreted code that can only be executed while holding the GIL, which means if a lot of threads want to run Python code in parallel they can potentially interfere with each other.

The alternative solution to running Python code in parallel is to use multiprocessing. We can create a separate process that runs the Python code. However, this introduces a new set of problems. Since the Python code is now running in a separate address space, we have to transfer data between the database process and the Python process.

Multiprocessing

On POSIX systems, we can perform multiprocessing using the `fork` system call. This system call creates a new process by cloning the existing process. The `fork` system call uses *copy-on-write* to copy the memory of the parent process [54]. Both the parent and child processes share all their memory pages until either of them writes to some memory page. When this happens, the child makes a copy of the memory page to which is written. As the main process already has the input columns in memory, and the input columns are read-only, we can use the `fork` system call to create a new process without copying the input columns.

However, this does not solve the problem of the output columns. We will need to transfer the output columns back to the main process using some form of interprocess communication. Since we are copying a large amount of data in bulk, using pipes or sockets is inefficient, as they have synchronization overhead. Instead, we copy the output columns to either a shared memory region, or a memory mapped file.

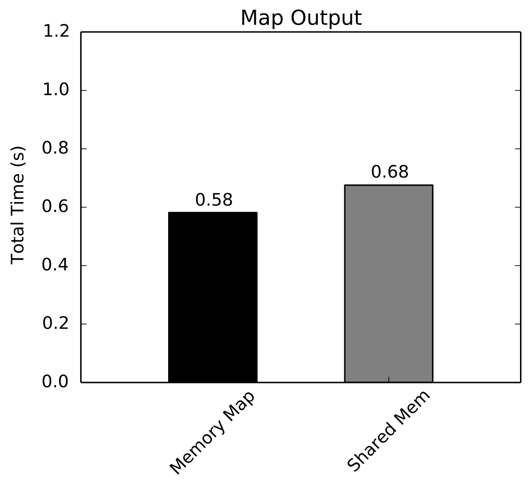


Figure 4.6: Copy 1GB of integers using interprocess communication.

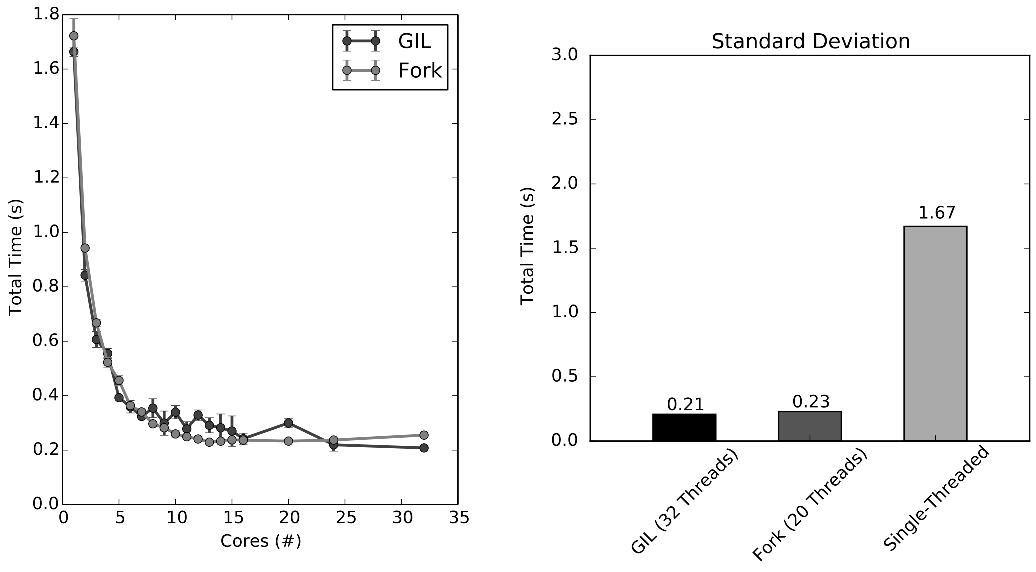
The advantage of multiprocessing is that Python code is executed completely in parallel. No matter how many threads or clients are connected, the processes do not influence each other. However, the price we pay for this is that we have overhead associated with interprocess communication. We have to copy the output columns along with some metadata back to the main process.

The advantage of running in a single process and working around the GIL is that we do not have to do extra copying of the output columns, as everything is running in the same address space. In addition, we do not have to do any context switching. However, if there are a lot of concurrent operations, performance could degrade as several threads fight for the GIL at the same time. This is especially problematic when the user-defined function uses operations that do not release the GIL, as then the entire function call blocks other user-defined functions from executing.

Output Conversion

In Figure 4.6, we tested which form of interprocess communication has the best performance. As we can see, there is a slight performance difference between shared memory and memory mapped files, although it is not very significant. What is significant is the cost we pay for the output conversion in the forked program. It takes over half a second to transfer the `1GB` of integers from the forked process back to the database server.

Standard Deviation Benchmark



(a) Scaling of function with cores. (b) Fastest computation time.

Figure 4.7: Parallel standard deviation of 1GB of integers.

In Figure 4.7, we tested how well the two approaches scaled with the amount of CPU cores used. In this benchmark, we computed an approximation of the standard deviation of a set of integers by computing the standard deviation of different chunks of the input column, and then averaging the resulting partial standard deviations. The standard deviation was computed using the `numpy.std` function, which releases the GIL while computing the standard deviation.

As we can see in Figure 4.7a, the two approaches stack up relatively evenly. They both scale well with the amount of cores being used. This is not very surprising. As we are only returning a single scalar value for each thread, the overhead of the multiprocessing approach is low. In addition, we are only calling a single `Python` function that releases the GIL while computing, thus the amount of GIL contention is not high in the non-forked approach.

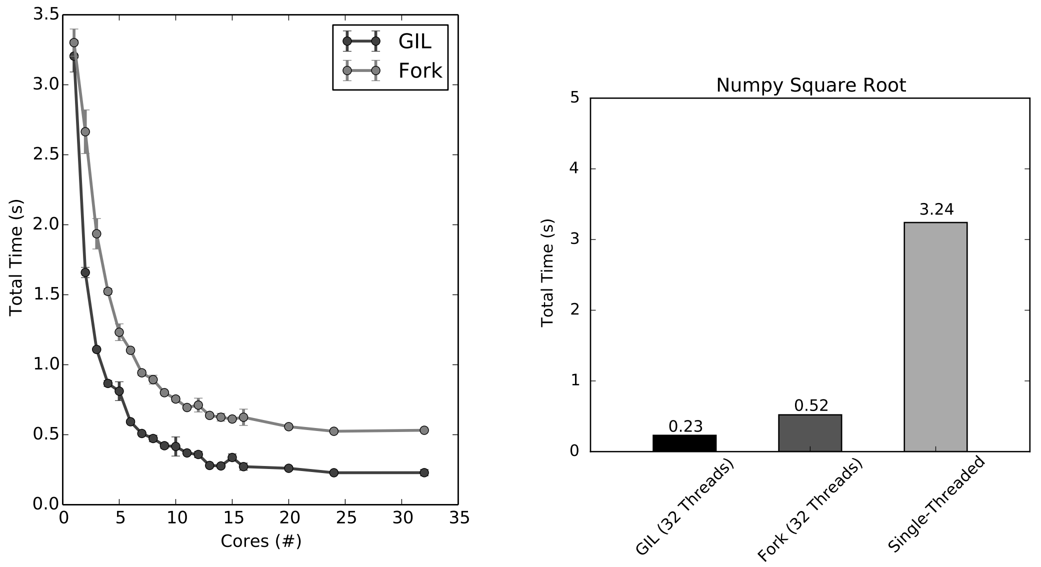
We do note that the non-forked approach has a higher deviation in processing time, while the multiprocessing approach is much more consistent. This is caused by the GIL contention of multiple threads, as the amount of contention for the GIL can vary considerably over several runs. If we are lucky, one thread will release the GIL just before another thread attempts to acquire it. However, if we are unlucky, two threads will attempt to acquire the GIL at the exact same time, forcing one thread to wait. As we can see, this can lead to significant variance in processing time, especially as the amount of cores used increases.

This benchmark shows that both approaches scale up relatively evenly. It should be noted that this benchmark is a best-case scenario for both approaches. There is almost no output to copy back to the main process, which favors the forked approach, and there is almost no GIL contention, which favors the non-forked approach.

NumPy Square Root Benchmark

In Figure 4.8, we tested how the two approaches stack up when the output is not a scalar value, but a significantly large column. In this benchmark, we compute the square root of every single value in a column of integers using our user-defined function. Thus the output column has the same size as the input column, which is `1GB` in this case.

We are using the vectorized operation `numpy.sqrt` to compute the square root for every element in the input column. This operation releases the GIL while operating, thus this is essentially the best case for the non-forked approach, hence we expect that to perform well. By contrast, the forked approach has to copy `1GB` of data back to the main process using interprocess communication, thus we expect that to perform significantly worse.



(a) Scaling of function with cores. (b) Fastest computation time.

Figure 4.8: Parallel `numpy.sqrt` of 1GB of integers.

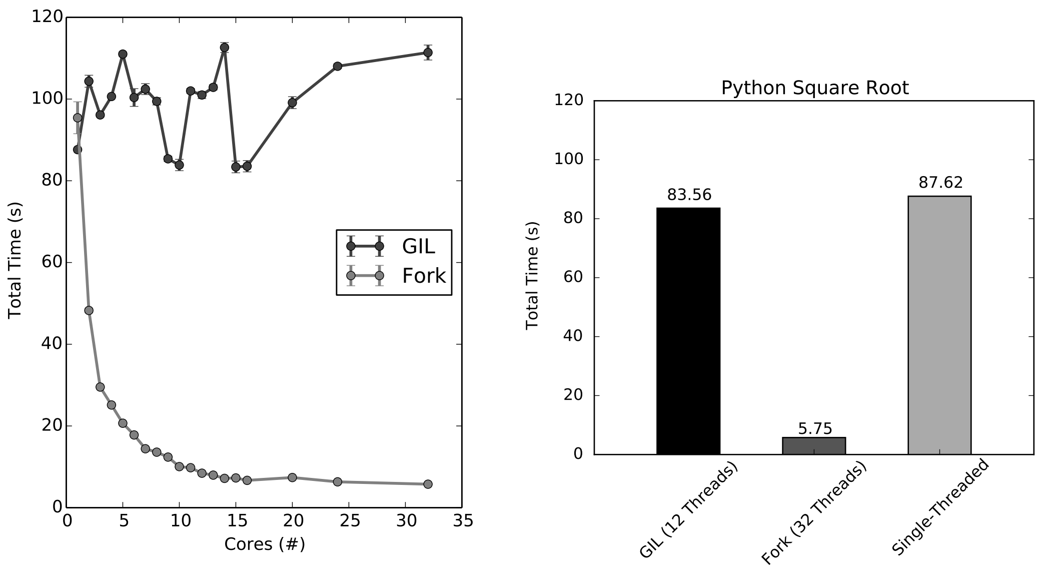
As we can see in Figure 4.8a, both approaches still scale relatively well with the amount of cores. However, we can see that the forked approach suffers from a consistent overhead that the non-forked approach does not have. This is caused by the forked approach having to copy `1GB` of output columns back to the main process. We can see that as the amount of cores increases, this copying overhead becomes the main bottleneck over the actual execution of the query.

As we can see in Figure 4.8b, the forked approach performs significantly worse than the non-forked approach in this scenario as expected. However, we still see a large benefit to performing this computation in parallel, as the forked approach is still over six times faster than the single-threaded approach.

Regular Python Functions Benchmark

In Figure 4.9, we tested how the two approaches compare when using regular Python functions that do not release the GIL, as opposed to NumPy functions that do. Just like in the previous benchmark, we are using the user-defined function to compute the square root of every single value in a `1GB` column of integers. However, unlike in the previous benchmark, we are now using the `math.sqrt` function to compute the square root, rather than the `numpy.sqrt` function. This function does not release the GIL while processing, hence we expect the non-forked approach to fall apart here, while we expect the forked approach to perform well.

As we can see in Figure 4.9a, the non-forked approach has jittery and unpredictable behavior because of the GIL. However, while the overall behavior seems inconsistent, the individual points do not have high standard deviation. Even though we have performed multiple measurements for each point, all the measurements are very similar. If lock contention and OS thread scheduling were the cause of the unpredictable execution times, we would expect high standard deviation in the individual points as well, rather than very predictable execution times.



(a) Scaling of function with cores. (b) Fastest computation time.

Figure 4.9: Parallel `math.sqrt` of 1GB of integers.

The reason for this odd behavior is that the GIL is not a mutex lock that a thread acquires and releases, as its name would suggest. Instead, the GIL functions more like an OS thread scheduler [55]. When there are multiple threads waiting for the GIL, the Python interpreter will switch between the currently executing thread after a fixed number of operations are performed. This thread switching happens after a Python thread performs hundreds of executions without releasing the GIL, as we are doing in this benchmark. As the Python interpreter switches after a fixed number of Python operations are performed, this switching is much more consistent than the thread switching of a standard OS scheduler. Which results in a low deviation in execution time for a given amount of threads.

However, this does not explain the large deviation between execution times on different amounts of cores. To understand that, we must look at the rather unique way in which the GIL switches between different execution contexts. When switching between different threads, rather than waiting until a separate thread obtains the GIL, the releasing thread immediately enters the GIL queue again and starts fighting for the GIL. This can lead to odd behavior, where the same thread continues executing its code but releases and reacquires the lock every few hundred function calls. This GIL thrashing significantly degrades performance when multiple threads are in heavy contention for the GIL, and can cause large variation in performance.

Interquery Parallelism

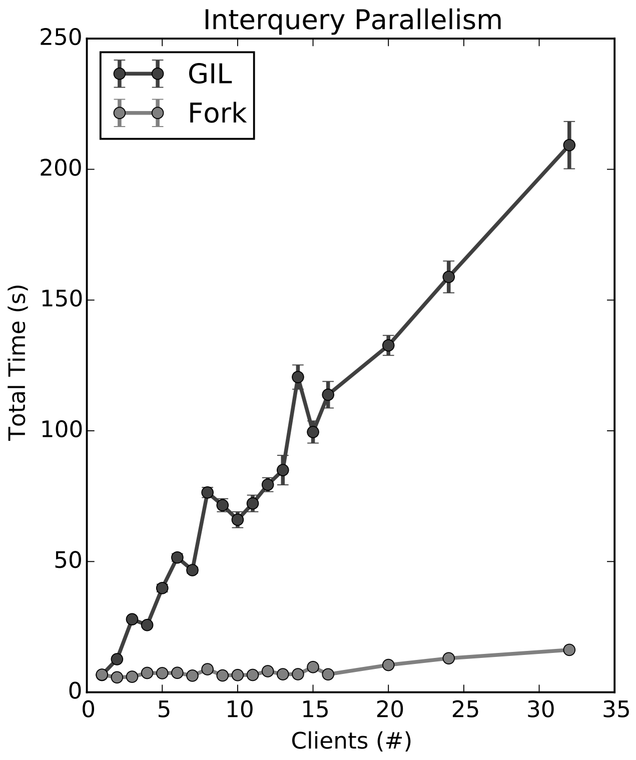


Figure 4.10: Square root computed by several clients in parallel.

In the previous benchmarks, we tested the intraquery parallelism of MonetDB/Python functions. However, we are also interested in the interquery parallelism of MonetDB/Python functions. In Figure 4.10 we tested how our two approaches scale with an increasing number of clients executing queries in parallel. The clients all sent a query to the database at the same time. The query computed the square root of `1GB` of integers using the non-vectorized `math.sqrt` function. The individual queries were set to only use a single core each. We then measured how long it took for the database to complete all the queries. In the best case, this would be a horizontal line, as then the clients would execute their queries completely in parallel without any overhead.

The results of the benchmark are shown in Figure 4.10. As we used the non-vectorized operation, the difference between the forked and non-forked approach vary wildly. For the forked approach, we can see that the total amount of time taken for all queries to complete slightly increases as additional clients are added until it reaches sixteen clients, which is the amount of physical CPU cores of the system. After that, the cost per client sharply increases, as the client threads cannot run in parallel anymore and have to alternate computations on the cores of the system.

The results are very different for the non-forked approach, as the GIL prevents almost all parallelization. The total time taken scales linearly with the amount of clients executing an operation at the same time. The last client would have to wait for all the other clients to finish the operation, leading to a very high variance in the response time.

Implementation and Implications

The non-forked approach stacks up relatively well when using vectorized NumPy functions that release the GIL. It even exceeds the performance of the forked approach when using functions that have a significant amount of output, as seen in Figure 4.8. However, it falls apart completely when non-vectorized operations are used, as seen in Figure 4.9 and Figure 4.10.

We might make the argument that non-vectorized operations should not be used, as they are highly inefficient. However, as we noted previously, NumPy does not offer vectorized string operations. In addition, the user might be more familiar with standard Python code, or use libraries that are written in Python. In all these scenarios, the non-forked approach does not parallelize at all.

Even when only using vectorized operations, the non-forked approach has a much higher deviation in execution time, as the different threads still interact with each other. In a lot of applications, such as when the response time of a query determines how long a user has to wait, we might prefer a more stable approach that has a consistent execution time, rather than one that is sometimes faster and sometimes slower.

Considering the advantage of using the non-forked approach is small even when using only vectorized operations, and the downside is so large when using non-vectorized operations, we have chosen to use the forked approach for both interquery and intraquery parallelism.

Parallel Processing of Aggregations

In the previous section, we described how we could process user-defined scalar functions in parallel. When an aggregate is used without the `GROUP BY` clause, it functions similarly to a scalar UDF. It receives the input columns as input, and must perform some operations on them and return an aggregated value.

However, when the `GROUP BY` clause is present, the user-defined aggregate must compute an aggregated value for every group. This introduces a new opportunity for parallelization, as we can compute the aggregate for every group in parallel. If we compute the aggregate separately for every group, however, we introduce significant overhead when there are a lot of unique groups, as we have to call the aggregate once for every group.

Rather than calling the aggregate once per group, we could call the aggregate function once, and give the user access to the group number of every row in an additional input column `aggr_group`. The user is then responsible for computing the different aggregate values for each group. This reduces the function call overhead as we only call the aggregate once, but we can no longer automatically parallelize the aggregate over the different set of groups. In addition, it pushes the responsibility of splitting the data and computing the aggregates onto the user.

To determine the performance difference between these two methods we have run a set of benchmarks comparing the two approaches. The performance difference between the two methods is primarily affected by the amount of groups. The parallel method suffers from function call overhead when there are a large number of groups, whereas the single function call method suffers less from the amount of groups, but does not offer the benefits of parallelization when there are a smaller number of groups.

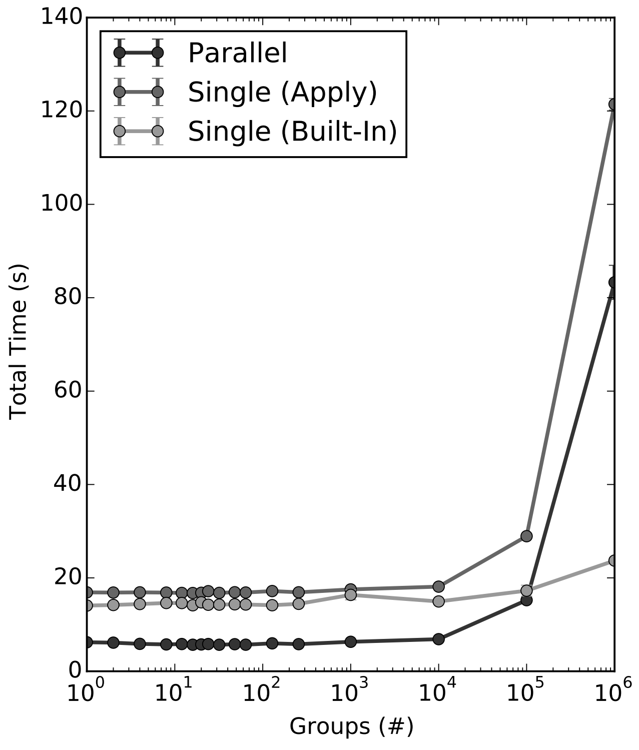


Figure 4.11: Compute the median of every group.

For the single function call approach, we use the Pandas library to compute the aggregates over the separate groups. The Pandas library supports two ways of aggregating over a set of groups. Either we can use one of the built-in Pandas aggregate functions, or we can supply our own Python function that will be called to compute the aggregate for each group.

For the benchmark, we computed the median of `1GB` of randomly generated integers spread evenly over n different groups, such that every group has `1GB`$/n$ integers. We expect the parallel approach to perform well when there are a small amount of groups, but we expect its performance to degrade as the amount of groups increases. We expect the Pandas aggregation using the built-in function to scale better with the amount of groups, as we are avoiding the large amount of function call overhead of the parallel approach. However, we expect that the Pandas aggregation where we apply our own Python function will scale just as badly as the parallel approach, as Pandas will have to call the Python function once for every group, meaning the overhead of the single Python function call per group is still present.

In Figure 4.11, the results of the benchmark are shown. We notice that the parallel execution performs significantly better than both the single function call aggregations at low group numbers. However, this is not because of the parallel execution of the aggregates. If the parallel execution was responsible, we would expect them to have similar performance when there is only a single group. Instead, the reason our parallel approach is more efficient is because pandas is less efficient in splitting the data.

As we expected, the performance of both the parallel approach and the applied-function approach degrade significantly as the amount of groups increases. When we have to call the Python function a million times, the interpreter overhead quickly eclipses the time it takes to perform the actual aggregations. The single-function approach using the built-in aggregate still has an increasing amount of overhead as the amount of groups increases, but it scales much better than the other approaches.

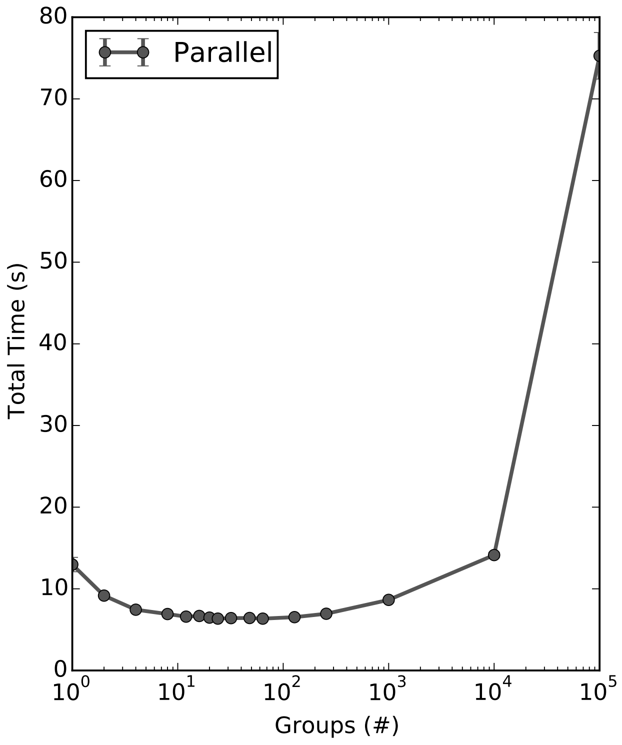


Figure 4.12: Compute the median of every group ten times.

While the parallel execution performs better with a lower amount of groups, we do not actually see a significant performance increase from the actual parallelization. Parallelization does not significantly improve the performance because computing the actual aggregation is not the bottleneck in this benchmark. Instead, most of the time is spent on collecting the actual groups and partitioning the data. We would expect parallelization to have a much bigger impact on performance if the aggregation is more expensive than computing the median.

We ran a second benchmark to measure the impact of parallelization for more expensive aggregates. In this benchmark, we computed the median of every group ten times, rather than once. As this aggregation takes significantly longer to compute, we expect the parallelization to provide a much bigger performance increase.

As we can see in Figure 4.12, the performance increase as the amount of groups increases is much more visible. The result is that the performance first improves as the amount of groups increases, because more parallelization becomes possible. After even more groups are added, the performance starts to degrade as the function call overhead becomes more and more significant.

Implementation and Implications

The parallel approach performs significantly better when there are a low amount of groups. It is especially efficient when an expensive aggregate is computed over a small amount of groups. It is also easier for the user to create parallel aggregations, as the system takes care of aggregating over the different groups. The user only has to create an aggregation that works for a single group and it will automatically work with the `GROUP BY` statement. However, as the performance decreases drastically when there are many unique groups, the single function approach could be significantly more efficient in some cases.

As such, we leave it up to the user to select one of these methods. Just like how they can specify whether or not their scalar user-defined function should be executed in parallel, they can specify whether or not the aggregate user-defined function can be executed in parallel when creating the aggregate.

Chapter 5

Evaluation

In this chapter we describe a set of experiments that we have run to test how efficient MonetDB/Python is compared to other database solutions and user-defined functions, as well as how efficient MonetDB/Python is to a range of other methods of loading data into Python.

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 (DIMM DDR3 1600 MHz (0.6 ns)) RAM. The machine uses the Fedora 20 OS, with Python version 2.7.5 and NumPy version v1.11.dev0. The source code used for the benchmarks can be found in Appendix A.

For each of the benchmarks, we ran the query four times. The result displayed in the graph is the mean of these four measured values. All benchmarks performed are hot tests unless stated otherwise. For the hot tests, we first ran the query twice to warm up the database prior to running the four measured runs. For the cold tests, we restarted the database and flushed the caches of the operating system prior to each of the measured runs.

5.1 Relational Databases

Relational Databases are among the most mature and most popular methods of storing data. There are a wide variety of different database management systems, and most of them support user-defined functions, at least in the language *C*.

5.1.1 In-Database Processing

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* and *C++* [8].

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 [10].

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* [56], however, there are wrappers that allow users to create 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.

Modulo Benchmark

For this microbenchmark, we will compute the modulo 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 rather 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.

In this benchmark, we compute the modulo of `1GB` of randomly generated 32-bit integers. The values of the integers are uniformly generated between the values 1 and 2^{31}. To ensure a fair comparison, every database 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.

The results of the benchmark are shown in Figure 5.1. 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 we would expect the row-store databases to have good performance.

Even when computing scalar functions, issuing multiple function calls 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 do not account for potential *null* values that could be in the database, and instead directly

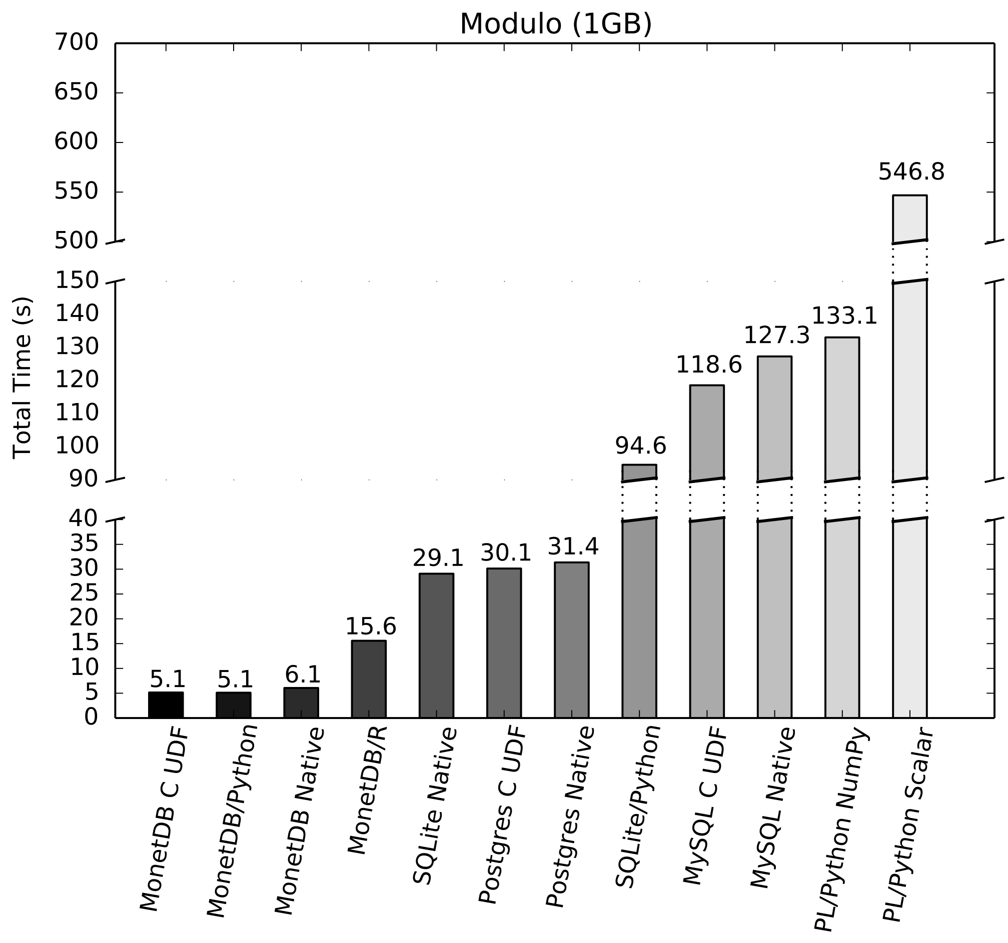


Figure 5.1: Modulo computation of 1GB of integers.

compute the modulo. This allows our user-defined functions to be faster than the built-in operators on all database systems.

Both SQLite/Python and PL/Python have very poor performance compared to the native modulo operator in their respective database. This is because Python is much slower than *C* when used in a *tuple-at-a-time* fashion. While the per-function overhead is already significant for *C* functions, this overhead is much larger for Python functions. As we have demonstrated in Chapter 4, Python is fast when used for vectorized processing in NumPy. However, vectorized processing does not fit naturally within the *tuple-at-a-time* processing model of SQLite and Postgres, which is why these functions use non-vectorized processing instead.

We implemented a vectorized version of the PL/Python UDF by loading the table into Python using the *loopback query* mechanism provided by Postgres. We then performed the vectorized computation using NumPy. As shown, this is much faster than the scalar PL/Python UDF, however, it is still not very fast. This is slow because the database is still loading the data into a set of Python objects one tuple at a time, and only then converting the data to a NumPy array.

MonetDB/R is slower than the *C* UDF in MonetDB because it makes a copy of both the `1GB` input column, and the `1GB` output column. This introduces a considerable amount of overhead. As the computation performed with the UDF is fast, most of the time is spent making copies of data that already exists in memory.

By contrast, MonetDB/Python is just as fast as the *C* UDF in MonetDB. This is because no time is spent converting the data from *C* to Python objects. The data is assigned to a NumPy array in *O(1)* time without copying any data. Then the vectorized NumPy operation is performed on the data, which is implemented in *C* and operates directly on a *C* array. The resulting NumPy array is then converted back into a *C* array in *O(1)* time without copying any data. In essence, we are calling a *C* function that operates on a *C* array with a small constant amount of interpreter and conversion overhead.

5.1.2 Database Connectors

While Figure 5.1 shows the performance of in-database processing, most databases do not support efficient Python UDFs. If users want to perform efficient in-database processing they must write *C* UDFs, which are difficult to create and require in-depth knowledge of the database kernel.

Instead of in-database processing using user-defined functions, users can use loose coupling of Python with the database server to use Python functions to process data from the database. The database server runs as a separate process from Python. Python then connects to the database through a SQL cursor connection. The user can then query the database through this connection. Python sends SQL queries to the database server, the database server then executes them and sends the results back to Python. This can be done either over the network, when the database server runs on a separate machine, or through interprocess communication when they run on the same machine.

MySQL provides connectivity with Python through the official MySQL Connector/Python module.

Postgres supports various Python connectors for Postgres. The most popular database connector and the one officially recommended by Postgres is *psycopg2*.

MonetDB provides connectivity with Python through the MAPI client.

SQLite does not provide a database connector, as there is no separate database server. Instead, SQLite runs embedded in Python. SQLite can be run in memory using the `:memory:` flag, or can be run from a database file.

Modulo Benchmark

Similar to the previous benchmark we performed, we will compute the modulo of a `1GB` set of randomly generated integers. However, instead of performing the computation in the database, we will load the data into Python using a database connector and then perform the computation in Python using the vectorized NumPy modulo function. In all benchmarks the database server was running on the same node as the Python connection, so no data had to be transferred over the network.

The results of the benchmark are shown in Figure 5.2. As we can see, the performance of the databases when using a database connector is significantly worse than when the computations are performed in the database. This is especially true for MonetDB and MySQL, who have poor performance when using the database connector to transfer the data. The reason for this is that both MonetDB and MySQL use sockets to transfer the individual rows from the database server to Python, even though Python and the database server are located on the same node.

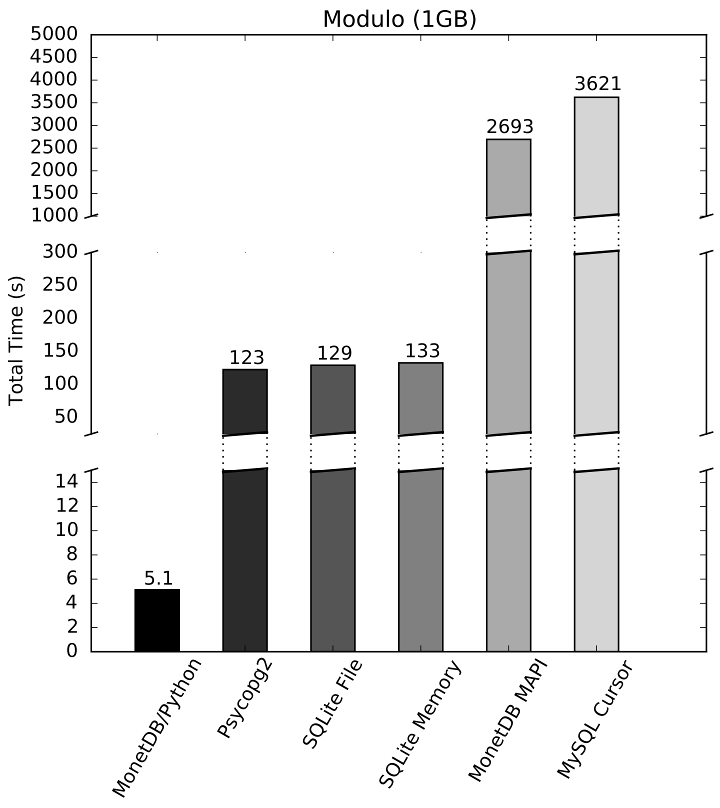


Figure 5.2: Modulo computation of 1GB of integers.

While sockets are slow, they are not *that* slow. Both MySQL and MonetDB have other issues that make the data transfer take a long time. MonetDB does not send binary data over the sockets. Instead, the integer values are first converted into encoded strings in the database server. Then the encoded data is sent over the sockets, and finally the encoded strings are converted back into integers. Not only does MonetDB spend a long time converting integers to strings and vice versa, the amount of data that has to be send over sockets is significantly larger in terms of bytes transferred, resulting in the data transfer taking significantly longer.

MySQL is slow because individual rows are send over the sockets instead of large binary blocks [57]. As an individual row only holds a single integer value in this benchmark, MySQL sends `1GB` of data over sockets in chunks of `4 bytes` each. In addition, after every packet sent MySQL waits for an acknowledgement that the packet was received. As a result, MySQL spends a very large amount of time sending packets back and forth between the server and the client.

Postgres performs much better. The Postgres connector still uses sockets, but it sends large chunks of binary data over the socket, leading to much better performance than either MySQL or MonetDB.

As SQLite runs directly embedded in Python, it does not have to deal with interprocess or network communication. However, loading the data into Python is still slow. The reason for this is that SQLite transfers the rows to Python one row at a time by constructing individual Python objects. This creates a lot of overhead, as constructing Python objects takes a very long time. In addition, the entire column must be copied to convert from the row-storage of SQLite to the column-storage of NumPy.

5.2 Alternative Data Storage in Python

There are various alternative solutions for data storage in Python. Naive solutions involve storing data in individual files, such as CSV files or NumPy binary files. However, this is not a scalable or flexible solution and will quickly become unmanageable for large data sets.

Other solutions include using specialized Python libraries for data loading and data storage. These come in various forms. Some are simple flat file databases, such as shelve [58]. Others are Python wrappers for rich file formats intended for storing complex data, such as PyFITS [59] and PyTables [60].

In this section, we will provide an in-depth analysis of all the available data storage solutions. How they work, their benefits and drawbacks, and how fast they are.

5.2.1 Flat File Storage

Flat file storage is the simplest form of storage available in Python. When using this form of storage, we keep the data in separate files, typically using standard formats.

The most common form of flat file storage is to have the data encoded in either a set of CSV, JSON or XML files, as these are highly portable formats that can be read by numerous applications. We can then load the files with the default Python modules `csv`, `json` or `xml`.

However, as these files are encoded in a specific format, reading the data from them is inefficient. We must first parse the file and convert the contents to the correct Python types. This is acceptable for smaller data sets, but the load times can grow out of control for larger data sets.

NumPy Binary Files

As an alternative to these portable formats, we can use the *NumPy* library to store *NumPy arrays* in binary format by using `numpy.save`. As these arrays are stored in binary format and we do not need to do any conversion loading them using `numpy.load` is very fast.

NumPy also supports memory mapping binary arrays using `numpy.memmap`. This uses the same binary representation as `numpy.save`, but it doesn't load the entire file into memory at once. Instead, it only loads parts of the file when they are requested. This is useful when we only need part of a larger column.

5.2.2 Hierarchical File Storage

Rather than storing individual data sets in individual files, which can quickly become unmanageable, hierarchical file storage stores the data in a single structured file.

HDF5 / FITs Files

The Hierarchical Data Format (HDF) [61] is a portable data format that stores data in a hierarchical structure. HDF5 is a very high performance format that can efficiently store and load arrays in a structured way.

The HDF5 format is supported in Python through the `h5py` [62] and PyTables modules [60]. The `h5py` module allows users to manipulate the HDF5 format at a low level, while the PyTables provides a higher level read/write interface that allows the user to load and store various Python objects and allows the user to index schemas for better performance

FITS files are a specialized data format built on top of HDF5 [59], used for storing and manipulating scientific images and associated meta data.

The HDF5 format is a high performance method of storing and retrieving data in a structured way, however, it does not provide functionality beyond data storage. In addition, when representing more complex relational data, we often introduce a lot of data redundancy that can be avoided by storing data in a relational database.

5.2.3 RDBMS Alternatives

In addition to the relational databases we have mentioned, there are several RDBMS alternatives. These do not only aim to provide persistent storage of data, but also aim to provide database principles such as concurrency control.

MongoDB is the most popular document-store database. Similarly to a relational database, it runs in a separate server process, and users can connect to it in Python through the `monary` client, which is optimized for data transfer to NumPy.

Castra is a column-store database implemented in Python. It runs embedded in Python, and stores objects using *pickling*. As a result, the performance of Castra is heavily dependent on the type of objects being stored

5.2.4 Percentile Benchmarks

To test the performance of each of these storage solutions and to compare it against the performance of MonetDB/Python and other databases, we performed a set of benchmarks. In the benchmark, we used each storage solution to load data set into Python, and then executed a Python function on the data. The total measured time is the loading time plus the execution time of the function.

The benchmark we have run computed the percentile of an integer column. The percentile is a generalized form of the median. Where the median is the center value of a set, such that 50% of the set is smaller than that point, and 50% of the set is larger than that point. The n^{th} percentile with $n \in [0, 100]$ is the value such that $n\%$ of the set is smaller than that point, and $100 - n\%$ of the set is bigger than that point. The median is equivalent to the 50^{th} percentile of a set.

The percentile was chosen for our microbenchmark for a number of reasons. It is a relatively simple computation, which means it is feasible to make our own implementation so we can also test user-defined functions in C. Most databases also have a built-in function that can be used to compute the percentile. In addition, the computation requires access to the entire column and cannot be calculated incrementally, unlike other aggregates such as `MIN`.

In this benchmark, we compute the 50th percentile of a table with a single column holding 1GB of 32-bit integer values, for a total of 250M rows. The actual values were randomly selected with a uniform distribution over the entire 32-bit integer range, ranging from -2^{31} to 2^{31}. The exact same set of values is used for every performance test.

We have performed two separate benchmarks. A hot benchmark, where the column has already been used prior to computing the data, so the data is already cached, and a cold benchmark, where the computation is performed right after the database has been started and all the caches have been flushed.

Hot Benchmark

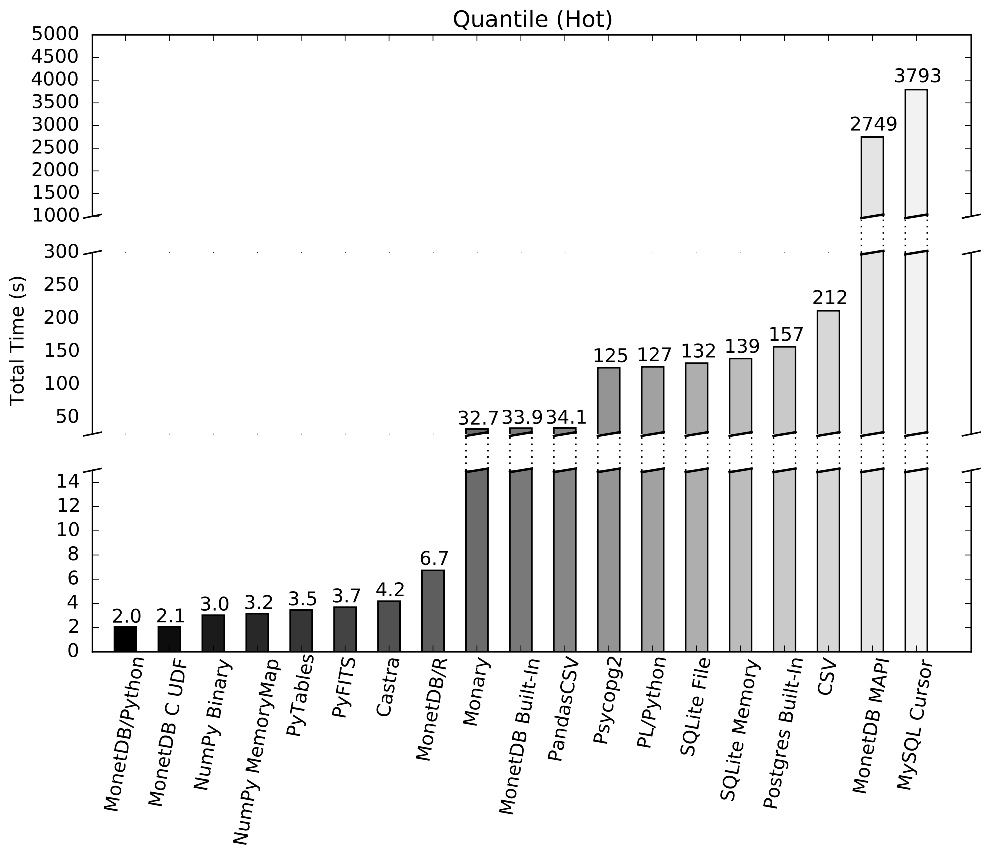


Figure 5.3: 50th percentile of 1GB of integers (Hot).

In Figure 5.3, the results of the hot percentile benchmark are shown. We can see that MonetDB/Python performs very well on the hot benchmark. As MonetDB is a main memory database, it holds the hot parts of the database in main memory. As the column we are computing the percentile on has recently been used, it is kept entirely in main memory, and does not have to be loaded from disk.

NumPy binary files have excellent performance as well. They load binary data directly from disk, and the operation system will cache the binary files, they can load the data into memory extremely fast without having to spent any time decoding the information in the files. The NumPy memory mapped files have slightly worse performance. As we are using all the data from the file, the memory mapping to only load part of it into memory does not offer any performance advantage over reading the entire file into memory. However, there is a slight overhead associated with memory mapping, which causes it to perform slightly worse.

Both PyTables and PyFITS have excellent performance. As they directly read a NumPy array from a high-performance HDF5 file that is cached by the operating system, they can very efficiently load the data into Python. These approaches are slower than the NumPy binary file, as the HDF5 file does incur some overhead costs and is less efficient than directly loading binary data from disk.

Castra also has good performance. This is because Castra uses pickling to store the NumPy array, which internally saves and loads the file using the NumPy binary `save` and `load` methods. However, it introduces some overhead causing it to be less efficient than NumPy binary files.

MonetDB/R also performs well, as the output is only a scalar value, the fact that it also copies the output values is not significant. However, its performance still suffers from having to copy the input data.

Monary uses sockets to transfer the data from MongoDB to Python, which has significant overhead. However, it directly constructs NumPy arrays from the transferred data. As it avoids tuple-at-a-time construction of Python objects, it is still efficient.

Surprisingly, the built-in quantile function of MonetDB performs significantly worse than both MonetDB/Python and the quantile C UDF. The reason for this is because the quantile implementation in MonetDB is inefficient. It sorts the entire column to obtain the quantile, which is much more costly than an efficient selection algorithm.

While the Pandas CSV reader is highly efficient and written in C, it still has to load and parse a non-binary CSV file and convert it into a NumPy array, causing it to perform badly.

The built-in Postgres quantile does not perform an efficient selection algorithm similarly to the built-in MonetDB quantile. Instead, it performs a full sort on the data. However, in addition to the inefficient algorithm used by Postgres, the *tuple-at-a-time* processing model incurs a significant performance hit as Postgres must first cycle through the individual rows and then copy them before the sort can take place.

Similarly to Monary, psycopg2 uses sockets to transfer the data from Postgres to Python. This transfer has a significant amount of overhead, which causes it to perform poorly. In addition, psycopg2 constructs individual Python objects for each of the integers, which costs a significant amount of extra time.

PL/Python uses the loopback query provided by Postgres to load the data into Python and perform the NumPy quantile operation. It is inefficient as Postgres iterates over the individual rows and converts each individual row into a set of Python objects, rather than directly converting the input to a NumPy array. SQLite is inefficient for the same reason.

The default Python CSV reader has significantly worse performance than the Pandas CSV reader. This is because the default CSV reader is written in Python, and converts the individual integers into Python objects.

MonetDB transfers non-binary data to the MonetDB MAPI client over a socket connection. MonetDB first converts the binary data to the non-binary format, then serializes the non-binary data over the socket, and then construct a set of

Python objects from the non-binary data. As a result, the MAPI client has poor performance.

The MySQL Python client is highly inefficient as well. The data is transferred over sockets to the MySQL client, however, the packages that are sent only contain the individual rows. In addition, MySQL requires a confirmation package for every sent package. This back and forth between the MySQL client and the MySQL server over every row causes poor performance.

Cold Benchmark

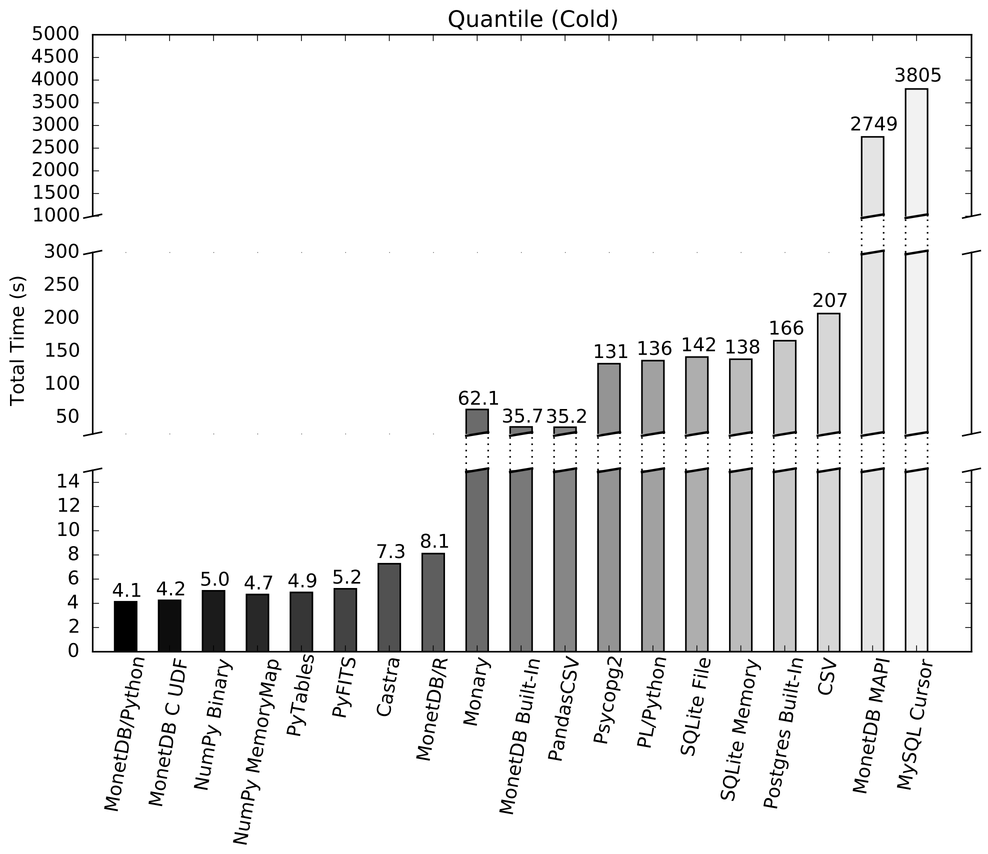


Figure 5.4: 50th percentile of 1GB of integers (Cold).

In Figure 5.4, the results of the cold percentile benchmark are shown. Compared to the hot benchmark, most data loading techniques see a serious performance drop.

While MonetDB/Python is still the fastest data technique, we do see that MonetDB/Python performs considerably worse when run on a cold dataset. Not only does MonetDB have to load the data from disk when the database has just been started, it also has to compile the query and construct the query plan again, which MonetDB caches after a query has been executed. The *C* UDF in MonetDB suffers from the same drastic performance drop.

The performance of the NumPy binary file loading degrades significantly when the NumPy binary file is not present in the system cache during the cold run. When the file is not cached, the system must load the binary file into memory from disk, causing significant performance degradation. The same behavior occurs for PyTables, PyFITS and Castra files.

Monary has an even more drastic drop in performance. MongoDB relies heavily on caching, and flushing the cache causes the performance of MongoDB to degrade heavily.

The Pandas CSV reader sees a slight drop in performance when the CSV has to be loaded from disk during the cold run, however, the bulk of the time is still spent on parsing the CSV file and converting the data into a NumPy array. This is even more true for the CSV loading using the built-in `csv` module, as significantly more time is spent parsing the CSV file here.

None of the database connectors see drastic performance changes between the cold and hot runs. This is because loading the data from disk is not the primary bottleneck for these connectors. Instead, they consistently have bad performance because of the processing model, the socket connection and the construction of individual Python objects. The fact that they must read the data from the disk does not have a significant impact on performance.

Chapter 6

Conclusion

In this section, we will draw our conclusions and discuss future work.

6.1 Conclusion

In this thesis, we have introduced the MonetDB/Python UDFs. By operating in an *operator-at-a-time* column-store database, there is almost no time spent transferring data to the scripting language. As a result, MonetDB/Python UDFs are highly efficient and not only faster than user-defined functions in other RDBMS, but also faster than other competing storage solution in Python.

We have tried to make MonetDB/Python UDFs as accessible as possible, and in doing so have solved issues other researchers had with user-defined functions. The user can use MonetDB/Python without requiring in-depth knowledge of the database kernel, and without having to compile and link the function to the database. As a result, MonetDB/Python functions are simple to create and simple to modify.

As MonetDB/Python functions work in a scripting language, there are no issues with potential memory leaks or unsafe memory access. The CPython garbage collector cleans up any objects used by the user in the user-defined function. This eliminates potential issues that can be caused by user-defined functions in C.

MonetDB/Python functions support polymorphic input, and supports automatic parallelization of functions over the cores of a single node. 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.

6.2 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.

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 greatly 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. Thus we cannot execute the function and look at the returned values to determine the column types. The solution proposed by Friedman et al. [12] 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.

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. [33], 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 3.1. 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 memory overhead.

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.

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.

Another interesting research direction is the query flow around user-defined functions. Currently, MonetDB/Python functions are treated as black boxes in query execution. However, queries involving MonetDB/Python functions could be optimized if we knew more about the computational complexity of the function. Determining this automatically is an extension of the halting problem, as if we could compute the exact run-time of a function, we would also know if the function would terminate. However, even though the halting problem is too difficult to solve, estimations could be made.

It has been suggested by Hellerstein et al. [31] and Chaudhuri et al. [32] to make the user specify the complexity of their function by making them fill in the cost per tuple. However, this can be very difficult to determine for the user and places the burden of optimization on them.

There has been some work by Crotty et al. [63] on automatically trying to estimate this information by looking at the actual compiled code. Alternatively, we could look at run-time statistics to try and determine the complexity of the functions, although this does require the user to use the function in an unoptimized data flow first.

Another interesting query flow problem relates to table-producing functions. 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, thus these heuristics do not work anymore.

We could solve this issue by introducing additional heuristics, either from user-provided information, by analyzing the function or gathered from statistics about the previous executions of the user-defined function. Alternatively, we could dynamically adapt the query plan based on the actual amount of columns.

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.

Bibliography

[1] E. F. Codd. “A Relational Model of Data for Large Shared Data Banks”. In: *Commun. ACM* 13.6 (June 1970), pp. 377–387. ISSN: 0001-0782. DOI: `10.1145/362384.362685`. URL: `http://doi.acm.org/10.1145/362384.362685`.

[2] Don Chamberlin. “SQL”. In: *Encyclopedia of Database Systems*. Ed. by Ling Liu and M. Tamer Özsu. Springer US, 2009.

[3] ISO. *ISO/IEC 9075:1992, Database Language SQL*. Tech. rep. July 1992. URL: `http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt`.

[4] Haixun Wang and Carlo Zaniolo. “User-Defined Aggregates in Database Languages”. English. In: *Research Issues in Structured and Semistructured Database Programming*. Ed. by Richard Connor and Alberto Mendelzon. Vol. 1949. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 43–60. ISBN: 978-3-540-41481-0. DOI: `10.1007/3-540-44543-9_4`. URL: `http://dx.doi.org/10.1007/3-540-44543-9_4`.

[5] Roger Magoulas John King. “2015 Data Science Salary Survey”. In: (Sept. 2015). URL: `http://duu86o6n09pv.cloudfront.net/reports/2015-data-science-salary-survey.pdf`.

[6] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. “Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications”. In: *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’98. Seattle, Washington, USA: ACM, 1998, pp. 343–354. ISBN: 0-89791-995-5. DOI: `10.1145/276304.276335`. URL: `http://doi.acm.org/10.1145/276304.276335`.

[7] Rakesh Agrawal and Kyuseok Shim. “Developing Tightly-Coupled Data Mining Applications on a Relational Database System”. In: *In Proc. of the 2nd Int’l Conference on Knowledge Discovery in Databases and Data Mining*. AAAI Press, 1996, pp. 287–290.

[8] URL: `http://dev.mysql.com/doc/refman/5.7/en/adding-udf.html`.

[9] URL: `http://docs.oracle.com/cd/B19306_01/server.102/b14200/functions231.htm`.

[10] URL: `http://www.postgresql.org/docs/9.3/static/external-pl.html`.

[11] *Extending HP Vertica*. URL: `https://my.vertica.com/docs/7.1.x/PDF/HP_Vertica_7.1.x_ExtendingHPVertica.pdf`.

[12] Eric Friedman, Peter Pawlowski, and John Cieslewicz. "SQL/MapReduce: A Practical Approach to Self-describing, Polymorphic, and Parallelizable User-defined Functions". In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1402–1413. ISSN: 2150-8097. DOI: 10.14778/1687553.1687567. URL: http://dx.doi.org/10.14778/1687553.1687567.

[13] *Data Science & BI: Salary & Skills Report.* URL: https://www.packtpub.com/skillup/data-salary-report.

[14] URL: http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext.

[15] Peter A. Boncz and Martin L. Kersten. "MIL Primitives for Querying a Fragmented World". In: *The VLDB Journal* 8.2 (Oct. 1999), pp. 101–119. ISSN: 1066-8888. DOI: 10.1007/s007780050076. URL: http://dx.doi.org/10.1007/s007780050076.

[16] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. "Column-stores vs. Row-stores: How Different Are They Really?" In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data.* SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 967–980. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376712. URL: http://doi.acm.org/10.1145/1376616.1376712.

[17] Stratos Idreos et al. "MonetDB: Two decades of research in column-oriented database architectures". In: *IEEE Data Eng. Bull* (), p. 2012.

[18] URL: https://www.monetdb.org/AboutUs.

[19] Andrew Eisenberg. "New Standard for Stored Procedures in SQL". In: *SIGMOD Rec.* 25.4 (Dec. 1996), pp. 81–88. ISSN: 0163-5808. DOI: 10.1145/245882.245907. URL: http://doi.acm.org/10.1145/245882.245907.

[20] Tobias Mayr and Praveen Seshadri. *Client-Site Query Extensions.* Tech. rep. Ithaca, NY, USA, 1998.

[21] Peter Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-pipelining query execution". In: *In CIDR.* 2005.

[22] Andrew Lamb et al. "The Vertica Analytic Database: C-store 7 Years Later". In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1790–1801. ISSN: 2150-8097. DOI: 10.14778/2367502.2367518. URL: http://dx.doi.org/10.14778/2367502.2367518.

[23] Volker Linnemann et al. "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. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1988, pp. 294–305. ISBN: 0-934613-75-3. URL: http://dl.acm.org/citation.cfm?id=645915.671798.

[24] Zhibo Chen and Carlos Ordonez. "Efficient OLAP with UDFs". In: *Proceedings of the ACM 11th International Workshop on Data Warehousing and OLAP.* DOLAP '08. Napa Valley, California, USA: ACM, 2008, pp. 41–48. ISBN: 978-1-60558-250-4. DOI: 10.1145/1458432.1458440. URL: http://doi.acm.org/10.1145/1458432.1458440.

[25] Zhibo Chen, Carlos Ordonez, and Carlos Garcia-Alvarado. "Fast and Dynamic OLAP Exploration Using UDFs". In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*. SIGMOD '09. Providence, Rhode Island, USA: ACM, 2009, pp. 1087–1090. ISBN: 978-1-60558-551-2. DOI: 10.1145/1559845.1559989. URL: http://doi.acm.org/10.1145/1559845.1559989.

[26] Carlos Ordonez. "Building Statistical Models and Scoring with UDFs". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 1005–1016. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247599. URL: http://doi.acm.org/10.1145/1247480.1247599.

[27] Qiming Chen, Meichun Hsu, and Rui Liu. "Extend UDF Technology for Integrated Analytics". English. In: *Data Warehousing and Knowledge Discovery*. Ed. by TorbenBach Pedersen, MukeshK. Mohania, and AMin Tjoa. Vol. 5691. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 256–270. ISBN: 978-3-642-03729-0. DOI: 10.1007/978-3-642-03730-6_21. URL: http://dx.doi.org/10.1007/978-3-642-03730-6_21.

[28] Qiming Chen et al. "Scaling-Up and Speeding-Up Video Analytics Inside Database Engine". English. In: *Database and Expert Systems Applications*. Ed. by SouravS. Bhowmick, Josef Küng, and Roland Wagner. Vol. 5690. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 244–254. ISBN: 978-3-642-03572-2. DOI: 10.1007/978-3-642-03573-9_19. URL: http://dx.doi.org/10.1007/978-3-642-03573-9_19.

[29] Michael Jaedicke and Bernhard Mitschang. "User-Defined Table Operators: Enhancing Extensibility for ORDBMS". In: *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. Ed. by Malcolm P. Atkinson et al. Morgan Kaufmann, 1999, pp. 494–505. ISBN: 1-55860-615-5.

[30] Carl-Fredrik Sundlöf. "In-Database Computations". MA thesis. Sweden: Royal Institute of Technology, Oct. 2010.

[31] Joseph M. Hellerstein and Michael Stonebraker. "Predicate Migration: Optimizing Queries with Expensive Predicates". In: *SIGMOD Rec.* 22.2 (June 1993), pp. 267–276. ISSN: 0163-5808. DOI: 10.1145/170036.170078. URL: http://doi.acm.org/10.1145/170036.170078.

[32] Surajit Chaudhuri and Kyuseok Shim. "Optimization of Queries with User-defined Predicates". In: *ACM Trans. Database Syst.* 24.2 (June 1999), pp. 177–228. ISSN: 0362-5915. DOI: 10.1145/320248.320249. URL: http://doi.acm.org/10.1145/320248.320249.

[33] Michael Jaedicke and Bernhard 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. Seattle, Washington, USA: ACM, 1998, pp. 379–389. ISBN: 0-89791-995-5. DOI: 10.1145/276304.276338. URL: http://doi.acm.org/10.1145/276304.276338.

[34] URL: http://db-engines.com/en/ranking.

[35] Daniel Abadi et al. “The Design and Implementation of Modern Column-Oriented Database Systems”. In: *Foundations and Trends in Databases* 5 (2013), pp. 197–280.

[36] Sándor Héman et al. “Positional Update Handling in Column Stores”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’10. Indianapolis, Indiana, USA: ACM, 2010, pp. 543–554. ISBN: 978-1-4503-0032-2. DOI: 10.1145/1807167.1807227. URL: http://doi.acm.org/10.1145/1807167.1807227.

[37] Milena Ivanova, Martin Kersten, and Fabian Groffen. “Just-in-time Data Distribution for Analytical Query Processing”. In: *Proceedings of the 16th East European Conference on Advances in Databases and Information Systems*. ADBIS’12. Pozna, Poland: Springer-Verlag, 2012, pp. 209–222. ISBN: 978-3-642-33073-5. DOI: 10.1007/978-3-642-33074-2_16. URL: http://dx.doi.org/10.1007/978-3-642-33074-2_16.

[38] URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[39] Guido van Rossum. URL: http://python-history.blogspot.nl/2009/01/brief-timeline-of-python.html.

[40] URL: http://www.kdnuggets.com/2014/08/four-main-languages-analytics-data-mining-data-science.html.

[41] URL: https://wiki.python.org/moin/PythonImplementations.

[42] URL: https://docs.python.org/2/reference/.

[43] URL: http://www.jython.org/.

[44] URL: http://ironpython.net/.

[45] URL: http://doc.pypy.org/en/latest/introduction.html.

[46] URL: https://docs.python.org/2/c-api/.

[47] URL: https://wiki.python.org/moin/GlobalInterpreterLock.

[48] S. van der Walt, S.C. Colbert, and G. Varoquaux. “The NumPy Array: A Structure for Efficient Numerical Computation”. In: *Computing in Science Engineering* 13.2 (Mar. 2011), pp. 22–30. ISSN: 1521-9615. DOI: 10.1109/MCSE.2011.37.

[49] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 2015-10-01]. 2001–. URL: http://www.scipy.org/.

[50] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[51] Wes McKinney. “Data Structures for Statistical Computing in Python”. In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[52] SymPy Development Team. *SymPy: Python library for symbolic mathematics*. 2014. URL: http://www.sympy.org.

[53] Michael A. Olson et al. “M.: Query Processing in a Parallel Object-Relational Database System, Data Engineering Bulletin”. In: *In Data Engineering Bulletin.* 1996, pp. 12–1996.

[54] Jonathan M. Smith and Gerald Q. Maguire Jr. “Effects of Copy-on-Write Memory Management on the Response Time of UNIX Fork Operations.” In: *Computing Systems* 1.3 (1988), pp. 255–278. URL: `http://dblp.uni-trier.de/db/journals/csys/csys1.html#SmithM88`.

[55] David Beazley. “Understanding the Python GIL”. In: Presented at PyCon 2010, 2010. URL: `http://www.dabeaz.com/python/UnderstandingGIL.pdf`.

[56] URL: `http://www.sqlite.org/c3ref/create_function.html`.

[57] URL: `https://dev.mysql.com/doc/refman/5.7/en/packet-too-large.html`.

[58] URL: `https://docs.python.org/2/library/shelve.html`.

[59] URL: `http://www.stsci.edu/institute/software_hardware/pyfits`.

[60] URL: `http://www.pytables.org/`.

[61] The HDF Group. *Hierarchical Data Format, version 5.* http://www.hdfgroup.org/HDF5/. 1997-NNNN.

[62] URL: `http://docs.h5py.org/en/latest/quick.html`.

[63] Andrew Crotty et al. “An Architecture for Compiling UDF-centric Workflows”. In: *Proc. VLDB Endow.* 8.12 (Aug. 2015), pp. 1466–1477. ISSN: 2150-8097. DOI: `10.14778/2824032.2824045`. URL: `http://dl.acm.org/citation.cfm?doid=2824032.2824045`.

Appendices

Appendix A

Micro Benchmarks

In this appendix, we present the SQL queries and any additional code that was used for the micro benchmarks performed throughout the thesis. But first, we present a table with the version numbers of each of the processes and modules that we benchmarked.

Name	Version
MonetDB	14abfa060341 (HG Branch)
MonetDB MAPI Client	11.19.3.2
Python	2.7.5
NumPy	1.11.dev0
Postgres	9.4.4
psycopg2	2.6.1
MySQL	5.7
SQLite	3 (Shipped with Python 2.7.5)
HDF5	1.8.16
PyTables	3.2.2.dev0
PyFITS	3.3
R	3.1.1
Pandas	0.16.2
Castra	0.1.6
MongoDB	3.07
Monary	0.4.0

Input Conversion

```
CREATE FUNCTION import_test(i INTEGER)
RETURNS DOUBLE
LANGUAGE PYTHON
{
    return 1.0
};
```

Listing A.1: Import test function.

In Listing A.1 the function used for conversion time testing is shown. We input a batch of integers and return a single constant value. This way, any Python computation time or output conversion time is negligible.

```
SELECT import_test(i) FROM integers;
```

Listing A.2: SQL query measured for input testing.

In Listing A.2, the usage of the `import_test` function is shown.

Input Conversion: Square Root

```
CREATE FUNCTION numpy_sqrt(i INTEGER)
RETURNS DOUBLE
LANGUAGE PYTHON
{
    val = numpy.sqrt(i)
    return 1.0
};
```

Listing A.3: Square root for NumPy array.

In Listing A.3, the function used for computing the square root in a vectorized fashion using a NumPy function is shown. We do not return the actual set of computed values, but return a scalar value. This is so the output conversion time is negligible and does not influence the benchmark.

```
CREATE FUNCTION pyobject_sqrt(i INTEGER)
RETURNS DOUBLE
LANGUAGE PYTHON
{
    import math
    val = [math.sqrt(x) for x in i]
    return 1.0
};
```

Listing A.4: Square root for PyObject.

In Listing A.3, the function used for computing the square root using the `math.sqrt` function is shown. This function operates on a single Python object, thus we have to loop over the input array.

String Input Conversion

```
CREATE FUNCTION import_test(i STRING)
RETURNS DOUBLE
LANGUAGE PYTHON
{
    return 1.0
};
```

Listing A.5: Input loading for string column.

In Listing A.5, the function used for testing the input conversion time of strings is shown.

String Operation: Lower

```
CREATE FUNCTION numpy_lower(i STRING)
RETURNS DOUBLE
LANGUAGE PYTHON
{
    val = numpy.core.defchararray.lower(i)
    return 1.0
};
```

Listing A.6: Lower operation for NumPy array

In Listing A.6, the function used for converting a NumPy string array to lowercase is shown. It uses the vectorized operation `numpy.core.defchararray.lower`.

```
CREATE FUNCTION pyobject_lower(i STRING)
RETURNS DOUBLE
LANGUAGE PYTHON
{
    val = [x.lower() for x in i]
    return 1.0
};
```

Listing A.7: Lower operation for `PyObject` array

In Listing A.7, the function used for converting a PyObject array to lowercase is shown. It uses the `str.lower()` operation on each of the individual python objects in the input to convert them to lowercase.

Standard Deviation

```
CREATE FUNCTION python_std(i INTEGER)
RETURNS DOUBLE
LANGUAGE PYTHON
{
    return numpy.std(i)
};
```

Listing A.8: Standard deviation for NumPy array

```
CREATE FUNCTION parallel_std(i INTEGER)
RETURNS DOUBLE
LANGUAGE PYTHON_MAP
{
    return numpy.std(i)
};
SELECT AVG(parallel_std(i)) FROM integers;
```

Listing A.9: Parallel standard deviation for NumPy array

Square Root

```
CREATE FUNCTION parallel_sqrt(i INTEGER)
RETURNS DOUBLE
LANGUAGE PYTHON_MAP
{
    return numpy.sqrt(i)
};
SELECT parallel_sqrt(i) FROM integers;
```

Listing A.10: Parallel sqrt for NumPy array

```
CREATE FUNCTION parallel_pysqrt(i INTEGER)
RETURNS DOUBLE
LANGUAGE PYTHON_MAP
{
    import math
    return [math.sqrt(x) for x in i]
};
SELECT parallel_pysqrt(i) FROM integers;
```

Listing A.11: Parallel PyObject sqrt.

Aggregations

```
CREATE AGGREGATE aggrmap(val INTEGER)
RETURNS INTEGER
LANGUAGE PYTHON_MAP
{
    return numpy.median(val)
};
SELECT groupcol, aggrmap(i) FROM integers GROUP BY groupcol;
```

Listing A.12: Parallel median computation over `GROUP BY`.

```
CREATE AGGREGATE pandasmedian(val INTEGER)
RETURNS INTEGER
LANGUAGE PYTHON
{
    import pandas as pd;
    df = pd.DataFrame.from_dict({'group': aggr_group, 'val': val});
    return numpy.array(df.groupby(['group'])['val'].median());
};
SELECT groupcol, pandasmedian(i) FROM integers GROUP BY groupcol;
```

Listing A.13: Median computation using `pandas` group by (built-in).

```
CREATE AGGREGATE pandasapply(val INTEGER)
RETURNS INTEGER
LANGUAGE PYTHON
{
    import pandas as pd;
    df = pd.DataFrame.from_dict({'group': aggr_group, 'val': val});
    return numpy.array(df.groupby(['group'])['val'].apply(numpy.
   median));
};
SELECT groupcol, pandasapply(i) FROM integers GROUP BY groupcol;
```

Listing A.14: Median computation using `pandas` apply.

Modulo

```
CREATE FUNCTION python_mod(i INTEGER)
RETURNS INTEGER
LANGUAGE PYTHON
{
    return numpy.mod(i, 100)
};
SELECT python_mod(i) FROM integers;
```

Listing A.15: MonetDB/Python modulo.

```
CREATE FUNCTION r_mod(i INTEGER)
RETURNS INTEGER
LANGUAGE R
{
    i %% 100
};
SELECT r_mod(i) FROM integers;
```

Listing A.16: MonetDB/R modulo.

```
SELECT i % 100 FROM integers;
```

Listing A.17: SQL modulo (MonetDB/MySQL/SQLite/Postgres).

```c
static char *
UDFBATmmod_(BAT **ret, BAT *src)
{
  BAT *bn = NULL;
  size_t i = 0;
  size_t cnt;
  int *result, *source;

  cnt = BATcount(src);

  bn = BATnew(TYPE_void, TYPE_int, cnt, TRANSIENT);
  BATseqbase(bn, src->hseqbase);

  result = (int*)bn->T->heap.base;
  source = (int*)src->T->heap.base;
  for(i = 0; i < cnt; i++) {
    result[i] = source[i] % 100;
  }
  BATsetcount(bn, cnt);

  *ret = bn;

  return MAL_SUCCEED;
}
```

Listing A.18: MonetDB C UDF for Modulo.

```c
Datum
mmod(PG_FUNCTION_ARGS)
{
    int32   arg = PG_GETARG_INT32(0);
    PG_RETURN_INT32(arg % 100);
}
```

Listing A.19: Postgres C UDF for Modulo.

```c
long long mmod(UDF_INIT *initid, UDF_ARGS *args,
              char *is_null, char *error)
{
    return (*((long long*) args->args[0])) % 100;
}
```

Listing A.20: MySQL C UDF for Modulo.

```sql
CREATE FUNCTION mmod(inp integer)
  RETURNS INTEGER
AS $$
return inp % 100
$$ LANGUAGE plpythonu;
```

Listing A.21: PL/Python Scalar UDF

```
CREATE FUNCTION mmod(inp integer)
  RETURNS INTEGER
AS $$
import numpy
cursor = plpy.cursor('SELECT i FROM integers')
ints = [x['i'] for x in cursor.fetchall()]
a = numpy.array(ints, dtype=numpy.int32)
val = numpy.mod(a, 100)
return 1
$$ LANGUAGE plpythonu;
```

Listing A.22: PL/Python NumPy UDF

Quantile

```
int quantile(int* x, size_t first, size_t last, size_t value) {
    size_t pivot, j, temp, i;

    if (first < last) {
        pivot = first;
        i = first;
        j = last;

        while(i < j) {
            while(x[i] <= x[pivot] && i < last)
                i++;
            while(x[j] > x[pivot])
                j--;
            if(i < j) {
                temp = x[i];
                x[i] = x[j];
                x[j] = temp;
            }
        }

        temp = x[pivot];
        x[pivot] = x[j];
        x[j] = temp;

        if (j > value)
           return quantile(x, first, j - 1, value);
        else if (j < value)
           return quantile(x, j + 1, last, value);
        else return x[j];
    }
    return x[first];
}
```

Listing A.23: *C* Quantile (Quickselect) used for MonetDB C UDF

```
SELECT quantile(i, 0.5) FROM integers;
```

Listing A.24: MonetDB built-in Quantile

```sql
SELECT percentile_cont(0.5) WITHIN GROUP(ORDER BY i) FROM integers;
```

Listing A.25: Postgres built-in Quantile

```sql
CREATE FUNCTION python_quantile(i INTEGER)
RETURNS INTEGER
LANGUAGE PYTHON
{
    return numpy.percentile(i, 50)
};
```

Listing A.26: MonetDB/Python Quantile UDF

```sql
CREATE FUNCTION r_quantile(i INTEGER)
RETURNS INTEGER
LANGUAGE R
{
    as.integer(quantile(a,0.5))
};
```

Listing A.27: MonetDB/R Quantile UDF

```sql
CREATE FUNCTION mquantile(inp integer)
  RETURNS INTEGER
AS $$
import numpy
cursor = plpy.cursor('SELECT i FROM integers')
ints = [x['i'] for x in cursor.fetchall()]
a = numpy.array(ints, dtype=numpy.int32)
val = numpy.percentile(a, 50)
return 1
$$ LANGUAGE plpythonu;
```

Listing A.28: PL/Python Quantile UDF

Appendix B

Input Conversion Challenges

In this section, we will describe some additional challenges with the data conversion from MonetDB to Python.

Huge Integers

MonetDB supports the huge integer data-type on certain systems. This type is a 128-bit integer, that can hold integer values between -2^{127} and 2^{127}. However, NumPy only supports 64-bit integer values, thus we must perform some conversion to make this type compatible with NumPy arrays.

The simplest solution would be to convert the huge integers to 64-bit floating point numbers, as 64-bit floating point numbers can hold values between -10^{308} and 10^{308}. However, this is at the cost of precision. While floating point numbers can represent very large values, they cannot represent them accurately.

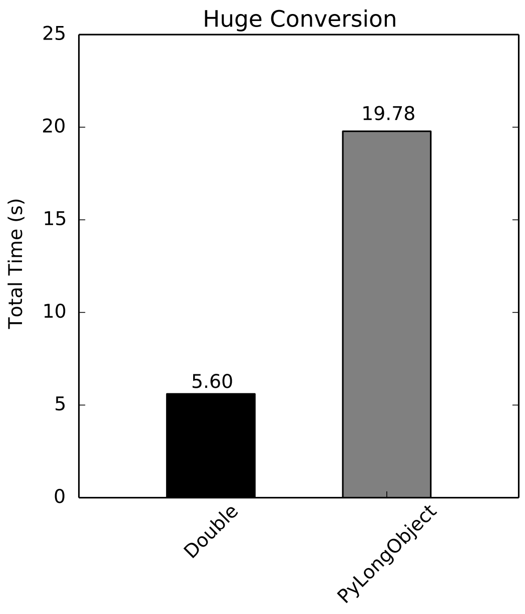
The alternative is to convert them to Python Objects of type `PyLongObject`. This Python type can hold arbitrarily large integral values. Internally, it is represented by an array of 32-bit integers. We would need four 32-bit integers (16 bytes) to accurately represent a huge integer. In addition, the `PyLongObject` has 24 bytes of overhead, for a total of 40 bytes of storage for every huge integer.

To determine the difference between `PyLongObject` and double conversion approaches, we have measured how long it took to convert to either of the representations. For this benchmark, we loaded in `4GB` of randomly generated huge integers. We measured the total conversion time, and how long operations took to perform on the set of huge integers.

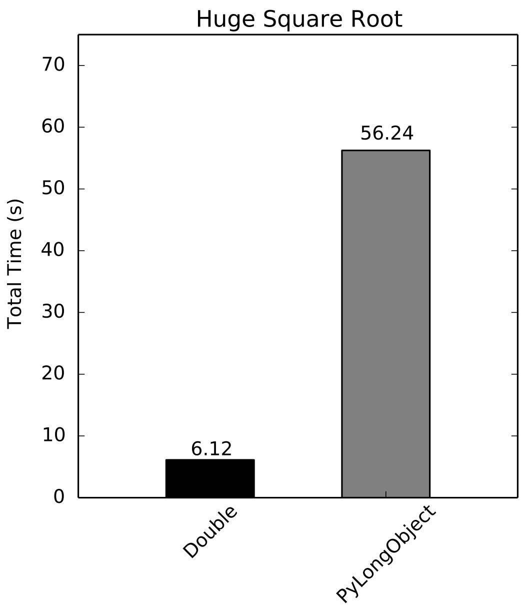
As we can see in Figure B.1a, it takes significantly longer to construct Python objects from huge integers. In addition, as we can see in Figure B.1b, any operations on the Python objects are significantly less efficient.

The question is, is the loss of precision worth this increase in performance? This depends on the user. It is possible for MonetDB to generate 128-bit integers from certain computations. In these cases, the user did not specifically request 128-bit integers and might not need exact precision, hence casting to a 64-bit floating point number might be preferable. However, if the user specifically requested a 128-bit integer, they likely did so for a reason and want to maintain the precision.

Since we cannot know the intentions of the user in every case, and whether or not the loss of precision poses a problem, we allow the user to specify which option to use. By default, we convert huge integers to floating point numbers, not only because this is significantly more efficient, but because a number of NumPy



(a) Construction Time of Huge Integers.



(b) Square Root Operation on Huge Integers.

Figure B.1: Benchmark: Different conversion methods for huge integers.

operations do not work properly on arrays of `PyLongObjects`. When this loss of precision happens, we show a warning alerting the user to the loss of accuracy.

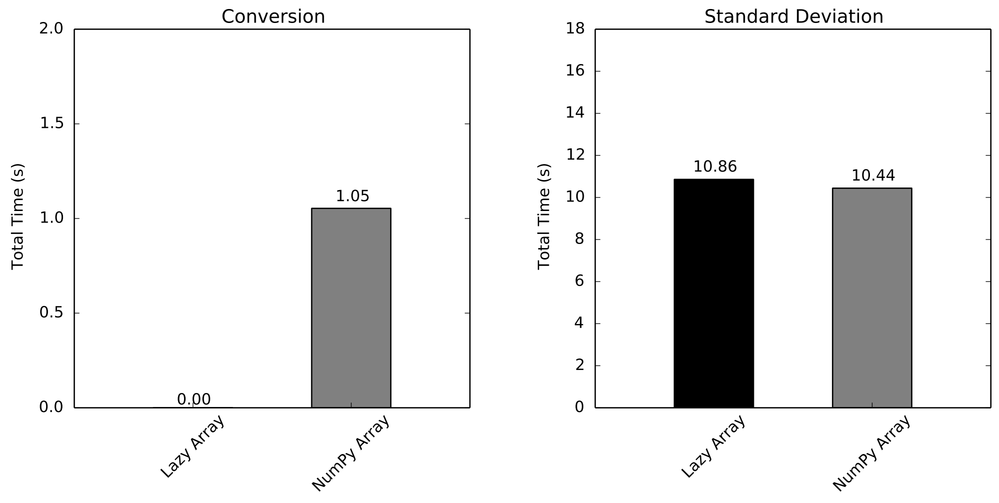
Lazy Loading

It is possible for users to pass columns to the Python function, but not use them. When these columns contain numeric columns without null values, this is not a problem, since the conversion time is constant. However, if these columns are string columns or they have null values the conversion involves scanning the input, which takes $O(n)$ time. If these columns are not used, we could be doing a significant amount of unnecessary work. We would prefer to load all the columns in a lazy fashion, and not perform any conversion until the columns are actually used.

If we were to load the data into such a lazy array, it must be indistinguishable from the actual NumPy array for the user. It must work properly with all the functions and libraries which work with NumPy arrays, and it must not degrade the performance of those operations.

We have attempted to implement such a lazy array by inheriting from the NumPy masked array class in Python. When the lazy array is constructed, a reference is set to the input BAT, but no conversion happens. When the data is accessed for the first time through Python, or when the `materialize` function is explicitly called, the BAT is converted to a NumPy array. Thus, when the data of the lazy array is never accessed, the data is never converted.

To benchmark how well our implementation performed, we performed a benchmark to compare our lazy array against a regular NumPy array on a set of integers with *null* values. First, we tested a function where the data in the array was never touched. In this case, the lazy array should never materialize the data, as it should



(a) Conversion time of unused column. (b) Compute standard deviation.

Figure B.2: Benchmark: A column holding 1GB of integers with *null* values.

only do this when the actual data is used. Next, we tested a benchmark where the input data was used. In this case, the lazy array and the NumPy array should have identical performance.

As we can see in Figure B.2a, when the column is not used no conversion occurs. When the column is actually used, as in Figure B.2b, the only difference between the NumPy array and the lazy array is that the lazy array performs the conversion at a later time. The performance is similar.

Unfortunately, while our lazy array properly delays the conversion of the data when the data is used in Python, it does not work with functions that directly access the internal *C* data of the array prior to materialization. As these functions directly access the *C* data, they bypass our check if the data is materialized, and try to access data that is not yet loaded.

We could use a type that does not inherit from the NumPy array type. This would force these functions to not look at the internal *C* data, but instead use the Python interface. This would fix this issue, but it would incur a significant performance penalty, as the reason these functions are so fast is that they directly use the internal *C* data.

The only way to implement lazy arrays that could be used transparently by the user and without impacting the performance of MonetDB/Python would be to modify the NumPy library and all the modules that directly use the NumPy array to properly account for these lazy arrays, and materialize the data before accessing it. However, as this would require us to modify countless libraries and ship them with MonetDB, we did not implement them in the final product.

Appendix C

Loopback Queries

MonetDB/Python functions can only take the input columns of a single table as input. If we want to use input columns from multiple tables, we have to join them together. However, this cannot always be done intuitively. Consider, for example, a table `classifiers` holding a classifier, and a table `testset` holding a set of data to be classified. We want to use the classifier in the table to classify the data in the data set. However, we cannot intuitively join the tables together, as they have no common join key. We could perform a cross product between the two tables, however, this is unintuitive and inefficient, as it would lead to a lot of data duplication.

To avoid unnecessary cross products, we support *Loopback Queries*. Loopback queries allow the user to query the database from within the MonetDB/Python function. In the previous scenario, the user can create `classify` UDF that takes the data from `testset` as input and loads the classifier from the `classifiers` table using a loopback query. An example of a MonetDB/Python function using a loopback query is given in Listing C.1.

```
CREATE FUNCTION classify(*)
RETURNS TABLE(id INTEGER, class INTEGER)
LANGUAGE PYTHON
{
    classifier = _conn.execute('SELECT * FROM classifiers')
    ...
};
```

Listing C.1: MonetDB/Python function using a loopback query.

The loopback query uses the same efficient input conversion we described in Section 4.2.1. The loopback query uses the same transaction context as the calling MonetDB/Python UDF. This is to ensure that the view of the database is consistent for the user, as they might have been making new tables or altering existing tables in the current transaction. Loopback queries do not necessarily have to be `SELECT` queries, they can contain arbitrary queries on the database, including `CREATE` or `UPDATE` queries. They could even contain other MonetDB/Python functions.

When the MonetDB/Python function is executed in the database process, executing these loopback queries is simple. We can simply execute the query using the query engine and use the input conversion mechanism we already have in place to convert the result to a set of Python objects. However, when the MonetDB/Python function is executed in a forked process, we cannot simply execute the query, as executing the query and loading data from the database in the forked process would ignore the database locks used by the main database process, which could cause serious issues when conflicts happen.

Instead, when a loopback query is used in a forked process, we ship the query back to the main process using interprocess communication, then execute the query in the database process and ship the result back to the forked process.

This introduces additional overhead when used in a forked process, as we have to copy the result of the query back to the forked process, which requires an *O(n)* scan over the data. However, the intention of loopback queries is not to load a significant amount of data into the UDF, but rather to load small objects that cannot easily fit into the SQL flow into the UDF, and for this purpose the overhead is not very significant.

Appendix D

Voter Classification

In Chapter 5, we performed a set of microbenchmarks that showed that MonetDB/Python has much better performance than alternative relational database solutions when used for simple operations. However, the main purpose behind MonetDB/Python is not to compute simple operations, such as the percentile, but rather to efficiently perform complex data analysis and data science in MonetDB by combining the power of SQL with the power of Python.

In this chapter, we will perform a more realistic benchmark. We will perform all the steps necessary for data classification from start to finish. We will load the data into the database, preprocess it, split the data into a train and test set, train a classifier using the train set and then test the resulting classifier on the test set, entirely using a combination of SQL and MonetDB/Python. As we did in the previous benchmark, we will test the performance of our system against alternative data storage solutions and other relational databases.

D.1 Problem Description

For this benchmark, we want to try and predict how people in North Carolina will vote (either Democrat or Republican) based on previous election results. For this, we have obtained two separate datasets that are publically obtainable from the North Carolina website.

The first dataset, *ncvoters*, contains the information about the individual voters. This is a dataset of `7.5M` rows with each row containing information about the person voting. There are 96 columns in total, but for simplicity we will only look at county, precinct, gender, ethnicity and age. Note that we do not know who the person actually voted for, as this information is not publicly available.

The second dataset, *precinct_votes*, contains the voting information for each precinct of the 2012 presidential elections. For each precinct, it contains how many people in each precinct voted for Obama (Democrat), and how many voted for Romney (Republican). This dataset has 2751 rows. This is publicly available information.

Our plan is to combine these two datasets to try to classify each of the voters. We know the voting records of a precinct, and we know in which precinct each person used, so we can try and guess who they voted for based on that information.

D.2 Implementation

First, we will have to combine the two datasets together. These are stored as two separate tables in our database, `ncvoters` and `precinct_votes`. This is where a database comes in handy, as we can simply JOIN the two tables on the county + precinct combination (we need this combination because there are multiple precincts with the same name, but in different counties). We can also immediately filter out every person that did not vote (this is stored in the column ncvoters.status).

```
CREATE TABLE ncvoters_joined AS
SELECT republican_percentage, county, precinct, sex, race,
    ethnicity, age
FROM precinct_votes
INNER JOIN ncvoters
ON ncvoters.precinct=precinct_votes.precinct AND ncvoters.county=
    precinct_votes.county
WHERE ncvoters.status='A' WITH DATA;
```

Listing D.1: Join ncvoter and precinct tables.

Preprocessing

Before we start classification, we first have to do some preprocessing. As we can see, all of these columns are stored as `STRINGS`. However, the classifier we will use only works with numerical values. This is where embedded python comes in handy. We can write a Python function that does the preprocessing for us.

We can use the `LabelEncoder` from the `sklearn` library to convert our `STRING` columns into `INTEGER` columns. The `LabelEncoder` assigns classes to every `STRING` value, so that every unique `STRING` value has a different class.

```
CREATE FUNCTION voter_preprocess(republican_percentage DOUBLE,
    county STRING, precinct STRING, sex STRING, race STRING,
    ethnicity STRING, age INT)
RETURNS TABLE(republican_percentage DOUBLE, county INT, precinct
    INT, sex INT, race INT, ethnicity INT, age INT)
LANGUAGE PYTHON
{
     from sklearn import preprocessing
     result_columns = dict()

     for key in _columns.keys():
         if _column_types[key] == 'STRING':
             le = preprocessing.LabelEncoder()
             le.fit(_columns[key])
             result_columns[key] = le.transform(_columns[key])
         else:
             result_columns[key] = _columns[key]
     return result_columns
};
```

Listing D.2: Function for preprocessing the data.

In this example we are using two special parameters, `_columns` and `_column_types`. `_columns` is a (key, value) dictionary that contains all the input columns. We use this dictionary so we can easily loop over all the input columns. `_column_types` is a (key, value) dictionary that contains the SQL types of the input columns; this is useful because the `numpy.object` column is not necessarily a `STRING` column.

We can then call this function on our joined table to create the preprocessed table.

```
CREATE TABLE ncvoters_preprocessed AS
SELECT *
FROM voter_preprocess( (SELECT * FROM ncvoters_joined) )
WITH DATA;
```

Listing D.3: Preprocess the data.

Stratified Sampling

Now we can finally begin our classification. For our classification, we will use a Random Forest Classifier. First, we will need to train our classifier on a train set, so we must first divide our dataset into a train and test set.

First, we will add unique ID numbers to our table so we can identify the rows.

```
ALTER TABLE ncvoters_preprocessed ADD ID INTEGER NOT NULL
    AUTO_INCREMENT;
```

Listing D.4: Add unique identifiers to rows.

Now we need to decide whether or not we want to include certain rows in the train set, or place them in the test set. Because the precincts are a vital factor in our classifier, we want to include people from every precinct. Instead of randomly selecting 5% of our data as train set, we will select 5% of the people from every individual precinct. This is again where Python comes in handy. In the Python function below we loop over every precinct, and then randomly select 5% of the people in that precinct and place them in the train set. This function returns a new table that holds a boolean value for every row that specifies whether or not it is included in the train set.

```
CREATE FUNCTION voter_split(precinct INT, id INT) RETURNS TABLE(id
    INT, train BOOLEAN)
LANGUAGE PYTHON
{
     count = len(id)
     indices = numpy.arange(count)
     train_indices = numpy.array([], dtype=numpy.int64)
     for p in numpy.unique(precinct):
         precinct_indices = indices[precinct==p]
         numpy.random.shuffle(precinct_indices)
         train_indices = numpy.append(train_indices,
    precinct_indices[:len(precinct_indices) / 20])
     train_set = numpy.zeros(count, dtype=numpy.bool)
     train_set[train_indices] = True
     return [id, train_set]
};
```

Listing D.5: Function for splitting train and test set.

```sql
CREATE TABLE train_set AS
SELECT *
FROM voter_split( (SELECT precinct, id FROM ncvoters_preprocessed)
   ) WITH DATA;
```

Listing D.6: Actually split the train and test set.

Training the classifier.

Now we can do the actual training of the data. As mentioned previously, we will use the RandomForestClassifier from sklearn. The only problem we have now is that we do not know the true classes of the voters. Instead, all we know is the percentage of people that voted Republican or Democrat in a given precinct. We will use this information to generate random classes for each person. We will randomly assign every voter a class of either 'Democrat' or 'Republican', weighted by the percentage of people that voted for a specific party in the precinct they live in. Consider a precinct in which 70% of the people voted for Romney and 30% voted for Obama. Each voter in that precinct has a 70% chance of being classified as a Republican voter, and a 30% chance of being classified as a Democrat voter.

After generating the classes we can do the actual fitting on our features, which are county, precinct, sex, race, ethnicity and age. After we have fitted the classifier, we need to be able to use it for predicting on our test set. For this, we need to be able to store our classifier somewhere and then access it in our voter predict function. For this, we use the `cPickle` module. Using the `cPickle` module, we can serialize Python objects into strings. This allows us to save the classifier directly in the database.

```
CREATE FUNCTION voter_train(republican_percentage DOUBLE, county
   INT, precinct INT, sex INT, race INT, ethnicity INT, age INT)
   RETURNS TABLE(cl_name STRING, cl_obj BLOB)
LANGUAGE PYTHON
{
    import cPickle
    count = len(county)
    from sklearn.ensemble import RandomForestClassifier
    clf = RandomForestClassifier(n_estimators=10)

    # randomly generate the classes
    random = numpy.random.rand(count)
    classes = numpy.zeros(count, dtype='S10')
    classes[random < republican_percentage] = 'Republican'
    classes[random > republican_percentage] = 'Democrat'

    del _columns['republican_percentage']

    # construct a 2D array from the features
    data_array = numpy.array([])
    for x in _columns.values(): data_array = numpy.concatenate((
   data_array, x))
    data_array.shape = (count, len(_columns.keys()))

    clf.fit(data_array, classes)

    return dict(cl_name="Random Forest Classifier", cl_obj=cPickle.
   dumps(clf))
};
```

Listing D.7: Function for training the classifier.

To train the classifier on the training set and store it in the `Classifiers` table we run the following query.

```
CREATE TABLE Classifiers AS
SELECT *
FROM voter_train(
 (SELECT *
  FROM preprocessed
  INNER JOIN train_set
  ON train_set.id=ncvoters_preprocessed.id
  WHERE train_set.train=true) )
WITH DATA;
```

Listing D.8: Training the classifier.

Testing the Classifier

Now we have successfully trained our classifier, all that remains for us is to use it to predict the classes of all the voters in the test set. Our prediction takes as input the features of the test set, and outputs the class for each row (either Democrat or Republican). We load the classifier from the database using a loopback query and use it to predict the classes of each of the rows of the test set. We can do this in SQL by grouping on the county and precinct.

```
CREATE FUNCTION voter_predict(...)
RETURNS TABLE(id INT, prediction STRING)
LANGUAGE PYTHON
{
    res = _conn.execute(
        "SELECT obj
         FROM Classifiers
         WHERE name='Random Forest Classifier'")
    classifier = cPickle.loads(res['obj'])

    result = dict()
    result['prediction'] = classifier.predict(_columns)
    result['id'] = id
    return result
};
```

Listing D.9: Function for using the classifier.

To run our prediction, we need to obtain the test set. We can do this by joining on the `train_set` table again.

```
CREATE TABLE predicted AS
SELECT *
FROM voter_predict(
 (SELECT *
  FROM preprocessed
  INNER JOIN train_set
  ON train_set.id=preprocessed.id
  WHERE train_set.train=false) )
WITH DATA;
```

Listing D.10: Predict on the test set.

We have successfully trained our classifier and used it to predict on the test set. All that's left is checking how accurate our classifier is. However, since we don't have access to the true classes of all our voters, this is not quite as easy as just counting the percentage of our predictions that were correct since we don't know which predictions were correct. Instead, we will look at the percentage of republican/democrat voters in each precinct and check how close our predicted percentage of democrat/republican voters is to the actual percentage of democrat/republican voters in each precinct. The results are shown in the table below.

Correct	Incorrect
1624	917

D.3 Performance Results

In this benchmark we are only loading in about 200MB of data, compared to the 1GB of data that we had to load for the percentile benchmark. In addition, we are performing significantly more complex calculations and are spending far longer in Python. As a result, the time spent loading the data is much less significant than it was in the previous benchmark.

However, even though this is the case, we still see that the existing relational database connectors spent a significant amount of time loading the data. Even

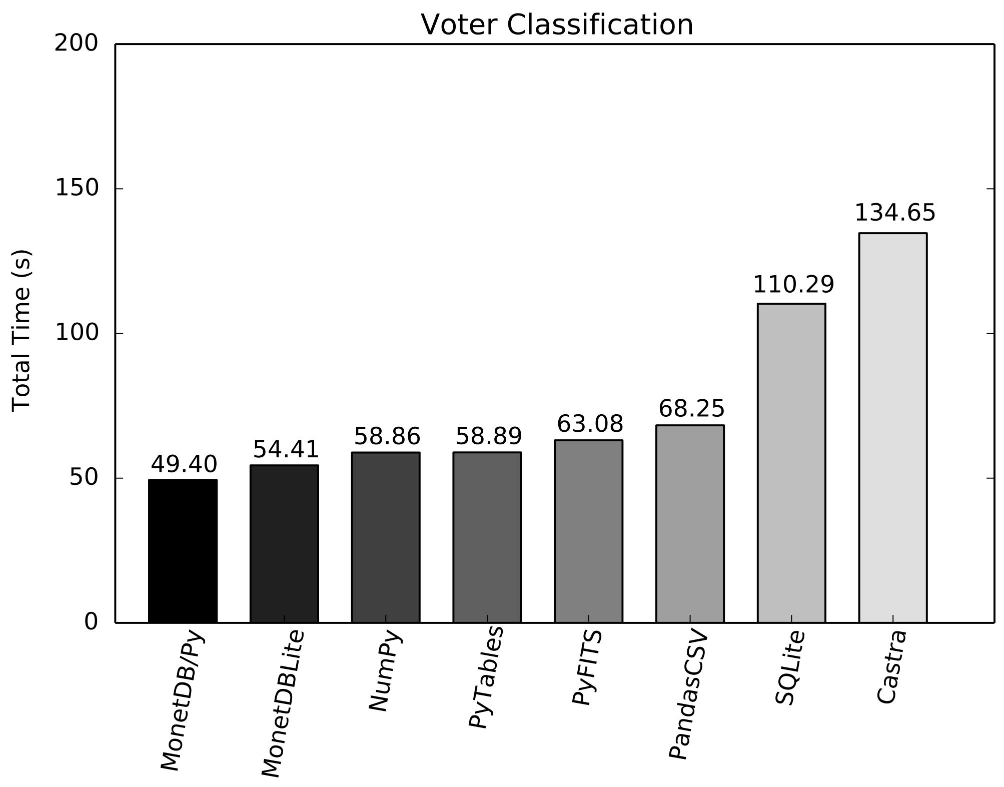


Figure D.1: Voter Classification using Random Forest Classifier.

though we have removed all superfluous columns from the table, the row-store databases do not perform well and take over twice as long to perform the data analysis operation. If we were to add the unused columns to the database, the performance of the row-store databases would degrade even further.