# Integrating Analytics with Relational Databases

Mark Raasveldt

# Integrating Analytics with Relational Databases

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus prof.mr. C.J.J.M. Stolker,
volgens besluit van het College voor Promoties
te verdedigen op dinsdag 9 juni 2020
klokke 11:15 uur

door

Mark Raasveldt
geboren te Leiderdorp
in 1992

**Promotiecommissie**

| | | |
|---|---|---|
| Promoter: | prof. dr. Stefan Manegold | (CWI & Leiden University) |
| Co-promoter: | dr. Hannes Mühleisen | (CWI) |
| Overige leden: | prof. dr. Aske Plaat | (Leiden University) |
| | prof. dr. Thomas Bäck | (Leiden University) |
| | dr. Mitra Baratchi | (Leiden University) |
| | prof. dr. Jens Dittrich | (Saarland University) |
| | prof. dr. Torsten Grust | (University of Tübingen) |

The research reported in this thesis has been carried out at the CWI, the Dutch National Research Laboratory for Mathematics and Computer Science, within the Database Architectures group.

The research reported in this thesis has been carried out under SIKS, the Dutch Research School for Information and Knowledge Systems.

This research is financially supported by the Dutch funding agency NWO, under project number 650.002.001 (the PROMIMOOC project), in collaboration with Tata Steel Ijmuiden, BMW Group Regensburg, Leiden University and Centrum voor Wiskunde en Informatica (CWI).

# Acknowledgments

When I was studying for my masters in university I always thought that I would never do a PhD. After all, you get paid less than working in industry and you need to work on theoretical research instead of solving practical problems. The reason that I opted to do a PhD anyway is because of Hannes, Stefan and the rest of the Database Architectures group. They showed me that not only can research be useful and practical, it can be amazingly fun and engaging as well. I have learned so much in my time here and am very thankful to each of the members of the DA group that have provided me with their knowledge and wisdom.

I am particularly indebted to Hannes, who took me in as a master student and has worked closely with me ever since. All of the days that we spent peer programming were extremely fun and informative. I would also like to give special thanks to my promotor Stefan. Firstly for giving me the opportunity of doing my PhD at the CWI, and secondly for being extremely kind and compassionate and creating such an accepting and amazing workplace. They say that how you experience your PhD depends entirely on your supervisor, and I had the best supervisors that I could wish for in Hannes and Stefan.

In my time at the CWI I have made many friends that have made my time there extremely pleasant. Pedro, Tim and Diego who I could always count on to have a good time and with whom I share many amazing memories. Thibault, who has taught me how to enjoy the pleasantries of life and how to write introductions. Abe, who has always impressed me with his math skills. Till, who was always ready to beat me in a game of table tennis. And finally, I would like to thank all the other people of the DA group for making my time there so special and amazing.

Finally, I would like to acknowledge my family and friends for their support. My mother and father - both also computer scientists - who have always supported me in doing whatever I wanted to do, even though I ended up following in their footsteps anyway. I would also like to thank my siblings, Maarten and Marieke. My close friends David and Dirk, and especially Ana who has always supported me.

# Contents

CHAPTER 1

# Introduction

## 1 The Rise of Data Science

Analyzing data in order to uncover conclusions, often referred to as "data science", is everywhere in todays world. In order to gain value from their data, nearly every large business has a data science branch or a team of data scientists looking to extract value from their data. But data analysis is not used only in the financial sector. It is also widely used in journalism, to aid the decision of government policies, in all branches of science and in many more areas.

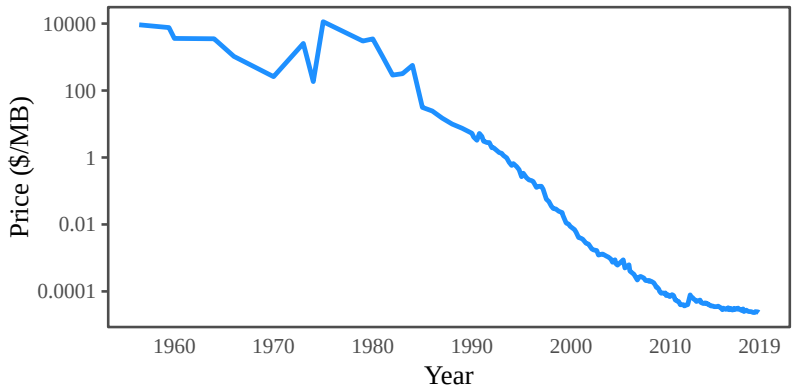



Figure 1-1: The cost of hard disks over time.

These developments are happening largely because of how cheap gathering, storing and analyzing large quantities of data have become. When we look back just sixty years, the IBM 350 disk storage unit was released. The IBM 350 could hold up to 3.75MB of data, and cost approximately 35000USD at the time. Today, we can buy a HDD with 1TB of storage for around 50USD. To put that into perspective, in 1960 the price of a Chevrolet Impala was around 3000USD. If the price of cars had fallen at the same rate as the price of hard disks, the newest model of Chevrolet would cost a mere $4.25 and could drive 200,000 times faster.

Not only has the cost of storing the data become so much cheaper, so has the cost of reading and processing that data. CPU processing speeds have improved orders of magnitude following Moore's Law, and RAM sizes have blown up. The phone in your pocket has over 10 times more high-speed RAM than the Cray-1 supercomputer had storage, and has significantly more processing power as well.

Looking at these numbers, it is no wonder that data science has become so ubiquitous. Analyzing large amounts of data has become very cheap and accessible even to small companies and individuals. Expensive supercomputers are no longer needed to store and analyze large amounts of data. Data science can be performed on cheap commodity hardware. Analyzing 10GB of data on a laptop is common place, and it is not unheard of to process 100GB or even 1TB of data on a desktop computer.

# 2 Tools of the Trade

While data science might appear like a brand new field, it is closer to a mixture of different fields. In particular, it is a combination of mathematics, statistics and computer science. Many of the techniques applied by data scientists are in fact techniques from the statistics field that can now cheaply be applied to large quantities of data because of technological advances.

Many of the tools that are used in data science have actually been designed and created by statisticians. An example of this is the R project for statistical computing [69]. The R language started as an open-source implementation of the

S language, a statistical language designed by John Chambers at Bell Labs. R was originally implemented by the statisticians Ross Ihaka and Robert Gentleman at Auckland for the purpose of teaching introductory statistics courses. It has grown into a tool that is used worldwide to perform statistical analysis, data classification and data visualization.

Another popular language for data science is Python, together with the support of the numeric python extensions NumPy [84], SciPy [48] and Pandas [64]. While Python itself does not have its roots in statistics, the numeric python extensions are based primarily on the APL family of languages, which includes S (the precursor of R), Fortran and MATLAB. These languages have all been designed primarily for use in numeric computing and statistics.

# 3 Data Science & Data Management

One of the consequences of the origins of these tools is that proper data management was never a first class citizen. Data management is largely treated as an afterthought in these tools. Typically, the data that is used for analyses is loaded from a data source into structures residing in memory and then kept around in memory. The tools do not support larger than memory data sets. Any management of that data is not handled by the tools themselves, and is left up to the user.

Data scientists typically opt to store the data in a set of flat files, as this is the most natural way of interacting with these tools. While flat file storage is simple when dealing with very small data sets that fit in individual files, it does not scale well. Flat file storage requires tremendous manual effort to maintain when the data sets grow in size. The files are also difficult to reason about because of the lack of a rigid schema, and it is difficult to share the data between multiple users. Furthermore, adding new data or modifying existing data is prone to corruption because of lack of transactional guarantees and atomic write actions provided by these tools.

All of the problems of flat file storage are not new problems. In fact, database management systems were created precisely to solve many of these problems. Modern

database management systems prevent data corruption through strong transactional guarantees and ACID properties, automatically manage data storage and make data easier to reason about by enforcing a rigid schema. In addition, the database management systems can perform efficient execution on larger-than-memory data, and allow safe concurrent access to the data.

However, despite the existence of database management systems, data scientists typically opt not to use them in conjunction with these analytical tools. This leads us to our main research problem:

**Research Problem** How can we facilitate efficient and painless integration of analytical tools and relational database management systems?

# 4 Our Contributions

In this thesis we work to answer the main research problem by investigating the different methods of combining relational database management systems and analytical tools. We consider the three separate methods of connecting analytical tools with RDBMSs: (1) client-server connections, (2) in-database processing and (3) embedded databases. For each of these methods, we examine the current state of the art and attempt to improve on it in both run-time efficiency and usability.

- **Client-Server Connections (Chapter 3).** We examine the client-server protocols of popular RDBMSs, and evaluate their effciency in the context of large-scale result export that is required to perform data analysis and machine learning on large data sets contained within these systems. Based on this analysis, we propose a new client-server protocol that handles these situations more efficiently and show its efficiency by implementing it in two open source RDBMSs.

- **In-Database Processing (Chapters 4 and 5).** We examine current methods of in-database processing in popular RDBMSs and improve on these methods by implementing a new method of in-database processing aimed at accelerating

in-database analytics: Vectorized UDFs. We implement these in MonetDB, a popular open-source RDBMS, and show how these UDFs can be effectively used to perform analytical workflows entirely within the RDBMS.

- **Embedded Database: MonetDBLite (Chapter 6).** We adopt the popular open-source RDBMS MonetDB to run as an embedded database inside analytical tools. We show how an embedded database can greatly increase usability of a database system, as well as show how the speed at which the analytical tool and the RDBMS can exchange data is greatly improved by embedding the database.
- **Embedded Database: DuckDB (Chapter 7).** Learning from our implementation of MonetDBlite, we identified the requirements and challenges of an embedded database system, and created our own RDBMS designed for being embedded from scratch: DuckDB. DuckDB fixes many of the deficiencies of MonetDBLite that were caused by the system being initially designed as a stand-alone server process.

# 5 Structure and Covered Publications

We present the background material necessary to understand this thesis in Chapter 2. We discuss the history of relational database management systems, and how they relate to the field of analytics, and we discuss the various ways in which database systems can be combined with stand-alone analytical tools.

In the subsequent chapters, we discuss the methods in which we aim to improve over the existing work. In Chapter 3, we describe our work on improving the client-server protocol, based on the following paper:

- **Don't Hold My Data Hostage - A Case For Client Protocol Redesign**
  Mark Raasveldt, Hannes Mühleisen
  43rd International Conference on Very Large Data Bases (VLDB 2017)

In Chapter 4 we discuss our work on extending user-defined functions for analytical use cases. This chapter is based on the following paper:

- **Vectorized UDFs in Column-Stores**
  Mark Raasveldt, Hannes Mühleisen
  28th International Conference on Scientific and Statistical Database Management (SSDBM 2016)

In Chapter 5 we discuss our work on embedding analytical workflows inside a database system. This chapter is based on the following paper:

- **Deep Integration of Machine Learning Into Column Stores**
  Mark Raasveldt, Pedro Holanda, Hannes Mühleisen and Stefan Manegold
  21st International Conference on Extending Database Technology (EDBT 2018)

In Chapter 6 we discuss our work on extending the MonetDB system into an embedded database system called MonetDBLite. This chapter is based on the (currently unpublished) paper:

- **MonetDBLite: An Embedded Analytical Database**
  Mark Raasveldt and Hannes Mühleisen

In Chapter 7 we discuss our work on creating the embedded database system DuckDB. This chapter is based on the following paper:

- **DuckDB: an Embeddable Analytical Database**
  Mark Raasveldt and Hannes Mühleisen
  ACM International Conference on Management of Data (SIGMOD 2019)
  Demonstration Track

# CHAPTER 2

# Background

As our goal is to improve the coupling of relational database management systems and analytical tools, it is clear that the existing techniques for combining external programs and RDBMS servers must be investigated.

In this chapter, we will describe existing techniques for combining external programs and RDBMS servers, and also provide the necessary background for understanding these techniques. In Section 1, we give a brief description of the history of RDBMS engines. In Section 2 we briefly describe the different types of RDBMS engines and the different physical storage models and database processing models they utilize. In Section 3 we discuss the different methods in which an external analytical program work in combination with a RDBMS. In Section 4 we describe the internal design of MonetDB, a popular open-source RDBMS that we have used as a test-bed for implementing a lot of the work in this thesis. In Section 5, we briefly describe the internal design of the CPython interpreter and the NumPy library, as we rely on these for the implementation of the vectorized user-defined functions described in Chapter 4.

# 1 Relational Database Management Systems

Database management systems have been around in some form or another for almost as long as computers themselves have been around. They solve the fundamental problem of manipulating and persistently storing data for future usage. This is a problem that is encountered by almost any application. Whether it be a bank manager, an online store or even a video game, they all require a method of persistently storing state, updating that state and reading back that state.

The most popular form of database management systems are relational database management systems. These types of systems allow users to interact with the data they store using languages based on relational algebra, pioneered by E.F. Codd [16] in 1970. In the relational model, data is organized in $n-ary\ relations$ where every row in the relation consists of $n$ different values. Data stored in this model can be stored into multiple relations, and combined at query time using the join operator ($\bowtie$).

The relational model offers two crucial advantages: (1) data can be stored in a normalized way, avoiding data duplication and improving data integrity, and (2) the way data is accessed is separated entirely from the physical way in which the data is organized, allowing for the engineers that create the database management system to have complete freedom in the way the data is represented on disk and the way in which it is accessed. This has allowed relational algebra to stay relevant even while storage methods, indexing algorithms and query execution models have changed.

After Codd's paper several languages popped up that were based on relational algebra. The clear winner, and the language used almost universally by relational database management systems today, is SQL [12] (Structured Query Language). SQL was initially developed in 1973 at IBM for use in System R [6] and was afterwards used in DB2 [91]. In the late 1970s it was adopted by Oracle [71], and it was standardized by the International Organization for Standardization (ISO) in 1987. Currently, SQL is the database language of choice for relational systems. It is supported by every major database vendor, and even many non-relational systems implement (limited) dialects of SQL as users are so familiar with it.

# 2 RDBMS Design

As relational algebra grants RDBMSs immense freedom in their underlying physical implementation, there have been many different proposed designs for RDBMSs. Each of the designs are catered towards different use cases, and have different advantages and disadvantages. In this section, we will discuss the most common trade-offs that are made in RDBMS design.

## 2.1 Workload Types

Before we discuss the types of RDBMS systems, we will describe the types of workloads that these systems are typically optimized for: OLTP workloads, OLAP workloads and hybrid workloads.

**On-Line Transactional Processing (OLTP)** workloads are focused on managing operational data for businesses. As an example of operational data, consider managing the in-stock items of a retailer, or updating account balances of a bank.

In OLTP workloads, there are many queries fired at the database concurrently. Individual queries are very simple and touch very few rows. In general, queries consist of either selecting, inserting, updating or deleting a single row. Queries that need to access data from a large subset of the database are (almost) never performed.

**On-Line Analytical Processing (OLAP)** workloads are focused on analyzing and summarizing the data stored inside a data warehouse. As an example of these analytical queries, consider for example generating business reports containing the sales of certain products over time, or the popularity of items in certain regions.

In OLAP workloads, there are relatively few queries fired at the database. However, the individual queries are very complex, and often touch the entire database. In these workloads, changes to the data in the form of inserts, updates or deletes happen in bulk (or might not even occur at all).

**Hybrid Workloads** consist of a mix of transactional statements and analytical statements. Typically, there is a high amount of small transactional statements fired at the system, mixed with the occasional reporting query.

## 2.2 System Types

**Disk-Based Systems.** When database systems were first created, computer systems were not equipped with much high-speed memory. When DB2 was originally released in 1987, the price of RAM was around 200USD per MB [55]. As such, these systems could not rely on a significant portion of the database fitting inside main memory. Instead, these systems were primarily designed for the database to reside on disk, with only a small portion of the data (that is currently being processed) residing in RAM. As the slow reading and writing speed of the hard disk was the primary bottleneck for these systems, they primarily considered how to optimize for minimizing disk access. These systems were primarily designed for OLTP workloads.

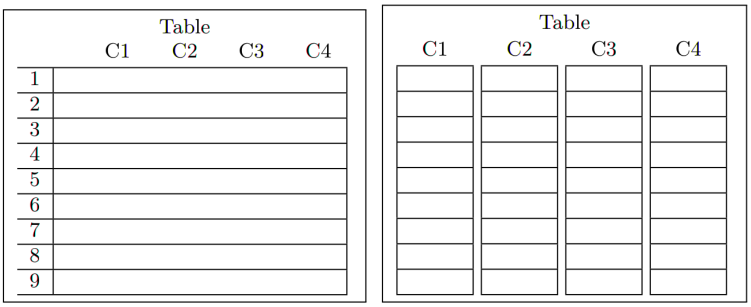
**Main-Memory Resident Systems.** When the prices of main memory fell and memory sizes grew, it became possible for the entire database (or at least the working set) to reside entirely in memory. As a result of the increasing memory sizes, it became possible to create systems optimized for main-memory resident data sets.

In systems optimized for main-memory, the disk no longer needs to be accessed at all for read-only queries, and data only needs to be written to disk for persistence purposes. As a result, these systems can achieve much faster speeds than the earlier systems that were bottlenecked by disk latency, but only if the system has sufficient memory to hold the working set. These systems have been designed for both OLTP, OLAP and hybrid workloads.

## 2.3 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 depending on the access pattern that is required by the query. The main decision in physical database layout is whether to horizontally fragment the data or to vertically fragment the data. These different physical layouts are visualized in Figure 2-1.

**Row Storage Databases** fragment tables horizontally. In this storage model, the data of a single tuple is tightly packed. The main advantage of this approach is




(a) Row-Store. (b) Column-Store.

Figure 2-1: Physical layout of row-store and column-store databases.

that operations on individual tuples are very efficient, as the data for a single tuple is tightly packed at a single location. The main drawback of this approach is that columns cannot be loaded individually from disk, as the values of a single column are surrounded by the values of the other columns. Because of this, unused columns in the table definition will affect query performance. When a query only operates on a subset of the columns of a table, the entire table must be loaded from disk regardless. This is especially relevant for OLAP-style queries that only touch a handful of columns in large tables with hundreds or even thousands of columns.

**Column Storage Databases** fragment tables vertically. In this storage model, the data of the individual columns is tightly packed. The advantages of this approach are two-fold: (1) the columns can be loaded and used individually, which means we do not need to load in any unused columns from disk, and (2) packing data of individual columns together leads to significantly better compression. The trade-off, however, is that reconstructing tuples is costly as the values of individual tuples are spread out over different memory locations. As a result, operations on individual tuples are expensive. As these types of operations are typically performed in OLTP workloads, column storage lends itself towards OLAP workloads.

## 2.4 Database Processing Models

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

between the database and the user-defined function. The processing model is closely related to the physical storage of the database.

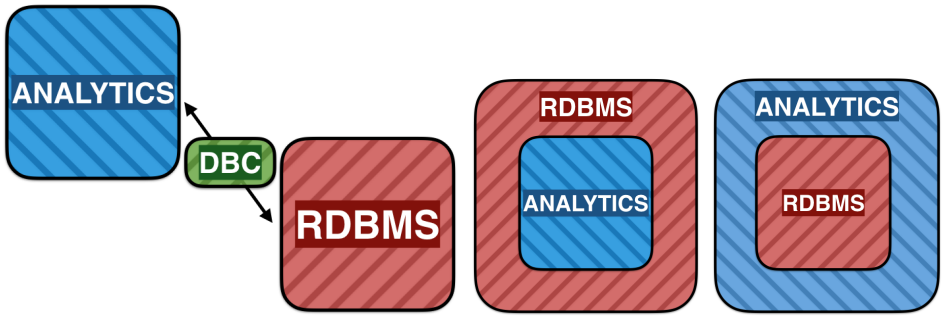
**Tuple-at-a-Time Processing** is the standard processing model used by most disk-based systems. In this processing model, the individual rows of the database are processed one by one from the start of the query to the end of the query.

The primary advantage of this processing model is that the system does not need to keep large intermediates in memory. In extremely low memory situations, processing queries in this fashion is often the only possibility. However, in situations where many rows are processed the tuple–at–a–time processing model suffers heavily from high interpretation overhead. This approach is used by PostgreSQL, MySQL and SQLite.

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

The main drawback of this processing model is the materialization cost of the intermediates of the operators. In the tuple-at-a-time processing model, a single tuple is processed from start to finish before the query processor moves on to the next tuple. By contrast, in the operator–at–a–time processing model, the operator processes the entire column at once before moving on to the next operator. Because of this, the intermediate result of every operator has to be materialized into memory so the result can be used by the next operator. As these intermediate results are the result of an entire column being processed they can take up a significant amount of memory. This approach is used by MonetDB.

**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 [8].




(a) Client-Server connection. (b) In-database processing. (c) Embedded database.

Figure 2-2: Different ways of connecting external programs with a database management system.

# 3 Database Connectivity

While RDBMSs are very powerful, SQL is not a general purpose language. As such, it is necessary for clients to write their actual application code in a different programming language and communicate with the RDBMS in order to exchange data between the application and the database management system.

As the focus of this work is on combining a RDBMS with analytical tools, we focus especially on users wanting to use analytical tools (e.g. Python or R programs) for the purpose of performing analysis on large amounts of data that reside in the RDBMS. Figure 2-2 shows the three main methods in which a relational database can be combined with an analytical tool. In this section, we will describe each of these methods and discuss how they operate from both a usability and a performance perspective.

## 3.1 Client-Server Connection

The standard method of combining a standalone program with a RDBMS is through a client-server connection. This is visualized in Figure 2-2a. The database server is completely separate from the analytical tool. It runs as either a separate process on the same machine or on a different machine entirely. The analytical tool communicates with the database server through a socket connection through an application programming

interface (API). After an initial authentication phase, the client can issue a query to the database server. The server will then execute the query. Afterwards, the result of the query will be serialized and written to the client over the socket. Finally, the client will deserialize the result.

The main advantage to this approach is that it is mostly database agnostic, as the standardized ODBC [32] or JDBC [24] connectors can be used to connect to almost every database. In addition, it is relatively easy to integrate into existing pipelines as loading from flat files can be replaced by loading from a database without having to modify the rest of the pipeline.

However, this approach is problematic when the client wants to run their analysis pipelines on a a large amount of data. The time spent on serializing large result sets and transferring them from the server to the client can be a significant bottleneck. In addition, this approach requires the full dataset to fit inside the clients' (often limited) memory.

## 3.2 In-Database Processing

In order to avoid the cost of exporting the data from the database, the analysis can be performed inside the database server. This method, known as in-database processing, is shown in Figure 2-2b.

In-database processing can be performed in a database-agnostic way by rewriting the analysis pipeline in a set of standard-compliant SQL queries. However, most data analysis, data mining and classification operators are difficult and inefficient to express in SQL. The SQL standard describes a number of built-in scalar functions and aggregates, such as *AVG* and *SUM* [43]. However, this small number of functions and aggregates is not sufficient to perform complex data analysis tasks [86].

Instead of writing the analysis pipelines in SQL, user-defined functions or user-defined aggregates in procedural languages such as C/C++ can be used to implement classification and machine learning algorithms. This is the approach taken by Hellerstein et al. [36]. However, these functions still require significant rewrites of existing analytical pipelines written in vectorized scripting languages. In addition, writing

user-defined functions in these languages require in-depth knowledge of the database internals and the execution model used by the database [14].

## 3.3 Embedded Databases

Both the client-server model and in-database processing require the user to maintain a running database server. This requires significant manual effort from the user, as the database server must be installed, tuned and continuously maintained. For small-scale data analysis, the effort spent on maintaining the database server often negates the benefits of using one.

Embedding the database system inside the client program, as shown in Figure 2-2c, is more applicable for these use cases. As the database can be installed and run from within the client program, maintaining and setting up the database is much simpler than with standalone database servers. As the database resides directly inside the client process, the cost of transferring data between the client and the database server is negated. The primary disadvantage of this solution is that only a single client can have access to the data stored inside the database server.

# 4 MonetDB

MonetDB is an open source column-store RDBMS that is designed primarily for data warehouse applications. In these scenarios, there are frequent analytical queries on the database, often involving only a subset of the columns of the tables, and unlike typical transactional workloads, insertions and updates to the database are infrequent and in bulk or do not occur at all. The core design of MonetDB is described in Idreos et al. [41]. However, since this publication a number of core features have been added to MonetDB. In this section, we give a brief summary of the internal design of MonetDB and describe the features that have been added to MonetDB since.

**Data Storage.** MonetDB stores relational tables in a columnar fashion. Every column is stored either in-memory or on-disk as a tightly packed array. Row-numbers for each value are never explicitly stored. Instead, they are implicitly derived from

their position in the tightly packed array. Missing values are stored as "special" values within the domain of the type, i.e. a missing value in an `INTEGER` column is stored internally as the value $-2^{31}$.

Columns that store variable-length fields, such as CLOBs or BLOBs, are stored using a variable-sized heap. The actual values are inserted into the heap. The main column is a tightly packed array of offsets into that heap. These heaps also perform duplicate elimination if the amount of distinct values is below a threshold; if two fields share the same value it will only appear once in the heap. The offset array will then point to the same heap entry for the rows that share the same value.

**Memory Management.** MonetDB does not use a traditional buffer pool to manage which data is kept in memory and which data is kept on disk. Instead, it relies on the operating system to take care of this by using memory-mapped files to store columns persistently on disk. The operating system then loads pages into memory as they are used and evicts pages from memory when they are no longer being actively used. This model allows it to keep hot columns loaded in memory, while columns that are not frequently touched are off-loaded to disk.

**Concurrency Control.** MonetDB uses an optimistic concurrency control model. Individual transactions operate on a snapshot of the database. When attempting to commit a transaction, it will either commit successfully or abort when potential write conflicts are detected.

**Query Plan Execution.** SQL is first parsed into a relational algebra tree and then translated into an intermediate language called MAL (Monet Assembly Language). MAL instructions process the data in a column-at-a-time model. Each MAL operator processes the full column before moving on to the next operator. The intermediate values generated by the operators are kept around in-memory if not too large, and passed on to the next operator in the pipeline.

Optimizations happen at three levels. High level optimizations, such as filter push down, are performed on the relational tree. Afterwards, the MAL code is generated and further optimizations are performed, such as common sub-expression elimination. Finally, during execution tactical decisions are made about how specific operations

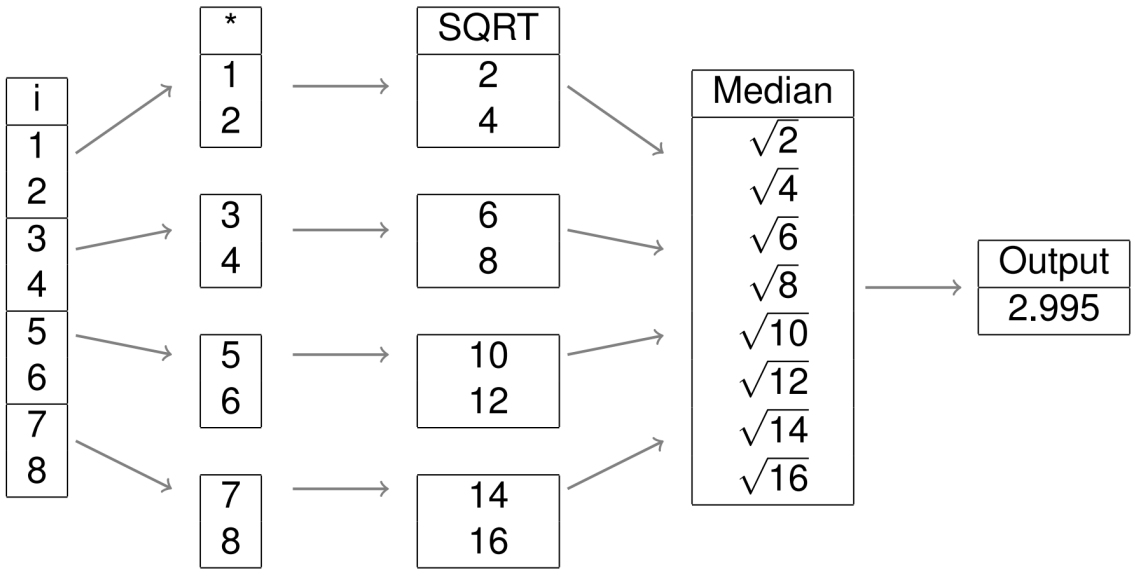



Figure 2-3: Parallel execution in MonetDB.

should be executed, such as which join implementation to use.

**Parallel Execution.** Initially, a sequential execution plan is generated. Parallelization is then added in the second optimization phase. The individual MAL operators are marked as either "blocking" or "parallelizable". The optimizers will alter the plan by splitting up the columns of the largest table into separate chunks, then executing the "parallelizable" operators once on each of the chunks, and finally merging the results of these operators together into a single column before executing the "blocking" operators. This is visualized in Figure 2-3 for the query `SELECT MEDIAN(SQRT(i * 2)) FROM tbl`.

The amount of chunks that are generated is decided by a set of heuristics based on base table size, the amount of cores and the amount of available memory. The database will attempt to generate chunks that fit inside main memory to avoid swapping, and will attempt to maximize CPU utilization. In addition, the optimizer will not split up small columns as the added overhead of parallel execution will not pay off in this case.

**Automatic Indexing.** In addition to allowing the user to manually build indices through the `CREATE INDEX` commands, MonetDB will automatically create indices during query execution.

Imprints [75] are a bitmap index that are used to assist in efficiently computing point and range queries. The bitmap index holds, for each cache line, a bitmap that contains information about the range of values in that cache line. They are automatically generated for persistent columns when a range query is issued on a specific column. They are then persisted on disk and used for subsequent queries on that column. Imprints are destroyed when a column is modified.

Hash tables are also automatically created for persistent columns when they are used in groupings or as join keys in equi-joins. These are also persisted on disk. Hash tables are destroyed on updates or deletions to the column. Unlike imprints, however, they are updated on appends to the tables.

**Order Index.** In addition to imprints and hash tables, MonetDB supports creation of a sorted index that is not created automatically. It must be created using the `CREATE ORDER INDEX` statement. Internally, the order index is an array of row numbers in the sort order specified by the user. The order index is used to speed up point and range queries, as well as equi-joins and range-joins. Point and range queries are answered by using a binary search on the order index. For joins, the order index is used for a merge join.

# 5 Python

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

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

Internally, CPython stores every variable as a `PyObject`. In addition to the value this object holds, such as an integer or a string, this object holds type information

and a reference count. As every `PyObject` can be individually deleted by the garbage collector, every Python object has to be individually allocated on the heap.

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

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

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

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

CHAPTER 3

# Database Client-Server Protocols

## 1 Introduction

In Chapter 2, we described how client-server protocols can be used to combine analytical tools with database servers. In this chapter, we dive further into the design of client-server protocols in modern database systems. Specifically, we focus on the manner in which result sets are (de)serialized and transported over a socket connection. While the performance of result set (de)serialization is irrelevant for smaller result sets, as the timing of the network will be dominated by the latency, the result set (de)serialization becomes very relevant when the client wants to export a large amount of data from the database system to a client program.

Figure 3-1 shows the impact that result set (de)serialization can have on query time. It displays the time taken to run the SQL query "`SELECT * FROM lineitem`" using an ODBC connector and then fetching the results for various data management systems. We see large differences between systems and disappointing performance overall. Modern data management systems need a significant amount of time to transfer a modest amount of data from the server to the client, even when they are

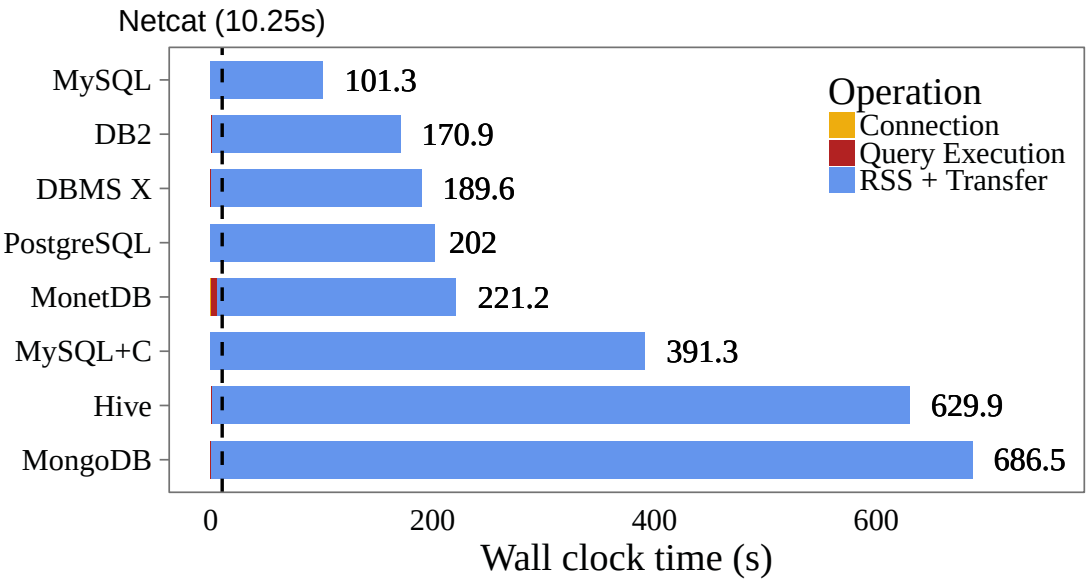



Figure 3-1: Wall clock time for retrieving the `lineitem` table (SF10) over a loopback connection. The dashed line is the wall clock time for netcat to transfer a CSV of the data.

located on the same machine.

## 1.1 Contributions

In this chapter, we investigate and benchmark the result set serialization methods used by major database systems, and measure how they perform when transferring large amounts of data in different network environments. We explain how these methods perform result set serialization, and discuss the deficiencies of their designs that make them inefficient for transfer of large amounts of data. We explore the design space of result set serialization and investigate numerous techniques that can be used to create an efficient serialization method. We extensively benchmark these techniques and discuss their advantages and disadvantages.Finally, we propose a new column-based serialization method that is suitable for exporting large result sets. We implement our method in the Open-Source database systems PostgreSQL and MonetDB, and demonstrate that it performs an order of magnitude better than the state of the art. Both implementations are available as Open Source software.

### 1.2 Outline

This chapter is organized as follows. In Section 2, we perform a comprehensive analysis of state of the art in client protocols. We analyze techniques that can be used to improve on the state of the art in Section 3. In Section 4, we describe the implementation of our proposed protocol and perform an extensive evaluation comparing our proposed protocol against the state of the art. We draw our conclusions in Section 5.

## 2 State of the Art

Every database system that supports remote clients implements a client protocol. Using this protocol, the client can send queries to the database server, to which the server will respond with a query result. A typical communication scenario between a server and client is shown in Figure 3-2. The communication starts with authentication, followed by the client and server exchanging meta information (e.g. protocol version, database name). Following this initial handshake, the client can send queries to the server. After computing the result of a query, (1) the server has to serialize the data to the result set format, (2) the converted message has to be sent over the socket to the client, and (3) the client has to deserialize the result set so it can use the actual data.

The design of the result set determines how much time is spent on each step. If the protocol uses heavy compression, the result set (de)serialization is expensive, but time is saved sending the data. On the other hand, a simpler client protocol sends more bytes over the socket but can save on serialization costs. The serialization format can heavily influence the time it takes for a client to receive the results of a query. In this section, we will take an in-depth look at the serialization formats used by state of the art systems, and measure how they perform when transferring large amounts of data.

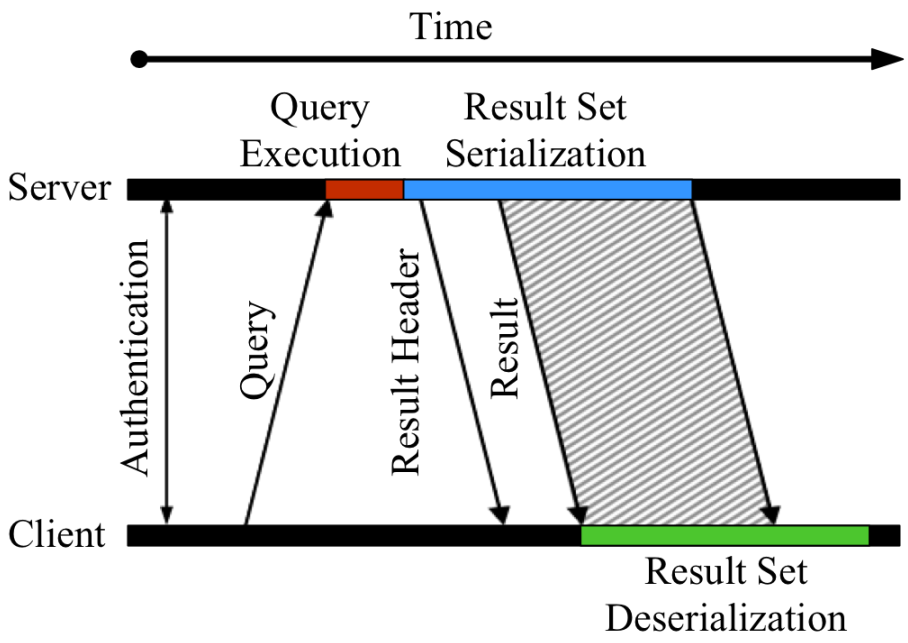



Figure 3-2: Communication between a client and a server

## 2.1 Overview

To determine how state of the art databases perform at large result set export, we have experimented with a wide range of systems: The row-based RDBMS MySQL [89], PostgreSQL [76], the commercial systems IBM DB2 [91] and "DBMS X". We also included the columnar RDBMS MonetDB [41] and the non-traditional systems Hive [81] and MongoDB [42]. MySQL offers an option to compress the client protocol using GZIP ("MySQL+C"), this is reported separately.

There is considerable overlap in the use of client protocols. In order to be able to re-use existing client implementations, many systems implement the client protocol of more popular systems. Redshift [35], Greenplum [25], Vertica [49] and HyPer [58] all implement PostgreSQL's client protocol. Spark SQL [5] uses Hive's protocol. Overall, we argue that this selection of systems includes a large part of the database client protocol variety.

Each of these systems offers several client connectors. They ship with a native client program, e.g. the `psql` program for PostgreSQL. This client program typically only supports querying the database and printing the results to a screen. This is

useful for creating a database and querying its state, however, it does not allow the user to easily use the data in their own analysis pipelines.

For this purpose, there are database connection APIs that allow the user to query a database from within their own programs. The most well known of these are the ODBC [32] and JDBC [24] APIs. As we are mainly concerned with the export of large amounts of data for analysis purposes, we only consider the time it takes for the client program to receive the results of a query.

To isolate the costs of result set (de)serialization and data transfer from the other operations performed by the database we use the ODBC client connectors for each of the databases. For Hive, we use the JDBC client because there is no official ODBC client connector. We isolate the cost of connection and authentication by measuring the cost of the `SQLDriverConnect` function. The query execution time can be isolated by executing the query using `SQLExecDirect` without fetching any rows. The cost of result set (de)serialization and transfer can be measured by fetching the entire result using `SQLFetch`.

As a baseline experiment of how efficient state of the art protocols are at transferring large amounts of data, we have loaded the `lineitem` table of the TPC-H benchmark [82] of SF10 into each of the aforementioned data management systems. We retrieved the entire table using the ODBC connector, and isolated the different operations that are performed when such a query is executed. We recorded the wall clock time and number of bytes transferred that were required to retrieve data from those systems. Both the server and the client ran on the same machine. All the reported timings were measured after a "warm-up run" in which we run the same query once without measuring the time.

As a baseline, we transfer the same data in CSV format over a socket using the netcat (`nc`) [33] utility. The baseline incorporates the base costs required for transferring data to a client without any database-specific overheads.

Figure 3-1 shows the wall clock time it takes for each of the different operations performed by the systems. We observe that the dominant cost of this query is the cost of result set (de)serialization and transferring the data. The time spent connecting to

the database and executing the query is insignificant compared to the cost of these operations.

The isolated cost of result set (de)serialization and transfer is shown in Table 3.1. Even when we isolate this operation, none of the systems come close to the performance of our baseline. Transferring a CSV file over a socket is an order of magnitude faster than exporting the same amount of data from any of the measured systems.

Table 3.1: Time taken for result set (de)serialization + transfer when transferring the SF10 lineitem table.

| System | Time (s) | Size (GB) |
|---|---|---|
| (Netcat) | (10.25) | (7.19) |
| MySQL | **101.22** | 7.44 |
| DB2 | 169.48 | 7.33 |
| DBMS X | 189.50 | 6.35 |
| PostgreSQL | 201.89 | 10.39 |
| MonetDB | 209.02 | 8.97 |
| MySQL+C | 391.27 | **2.85** |
| Hive | 627.75 | 8.69 |
| MongoDB | 686.45 | 43.6 |

Table 3.1 also shows the number of bytes transferred over the loopback network device for this experiment. We can see that the compressed version of the MySQL client protocol transferred the least amount of data, whereas MongoDB requires transferring ca. six times the CSV size. MongoDB suffers from its document-based data model, where each document can have an arbitrary schema. Despite attempting to improve performance by using a binary version of JSON ("BSON" [42]), each result set entry contains all field names, which leads to the large overhead observed.

We note that most systems with an uncompressed protocol transfer more data than the CSV file, but not an order of magnitude more. As this experiment was run with both the server and client residing on the same machine, sending data is not the main bottleneck in this scenario. Instead, most time is spent (de)serializing the result set.

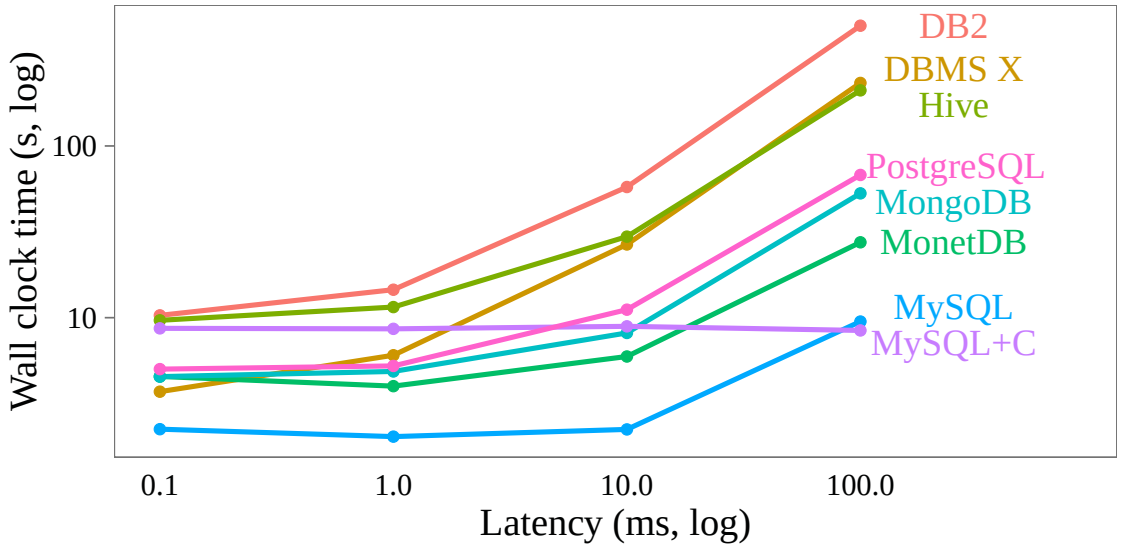



Figure 3-3: Time taken to transfer a result set with varying latency.

## 2.2 Network Impact

In the previous experiment, we considered the scenario where both the server and the client reside on the same machine. In this scenario, the data is not actually transferred over a network connection, meaning the transfer time is not influenced by latency or bandwidth limitations. As a result of the cheap data transfer, we found that the transfer time was not a significant bottleneck for the systems and that most time was spent (de)serializing the result set.

Network restrictions can significantly influence how the different client protocols perform, however. Low bandwidth means that transferring bytes becomes more costly; which means compression and smaller protocols are more effective. Meanwhile, a higher latency means round trips to send confirmation packets becomes more expensive.

To simulate a limited network connection, we use the Linux utility `netem` [39]. This utility allows us to simulate network connections with limitations both in terms of bandwidth and latency. To test the effects of a limited network connection on the different protocols, we transfer 1 million rows of the `lineitem` table but with either limited latency or limited bandwidth.

**Latency.** An increase in latency adds a fixed cost to sending messages, regardless

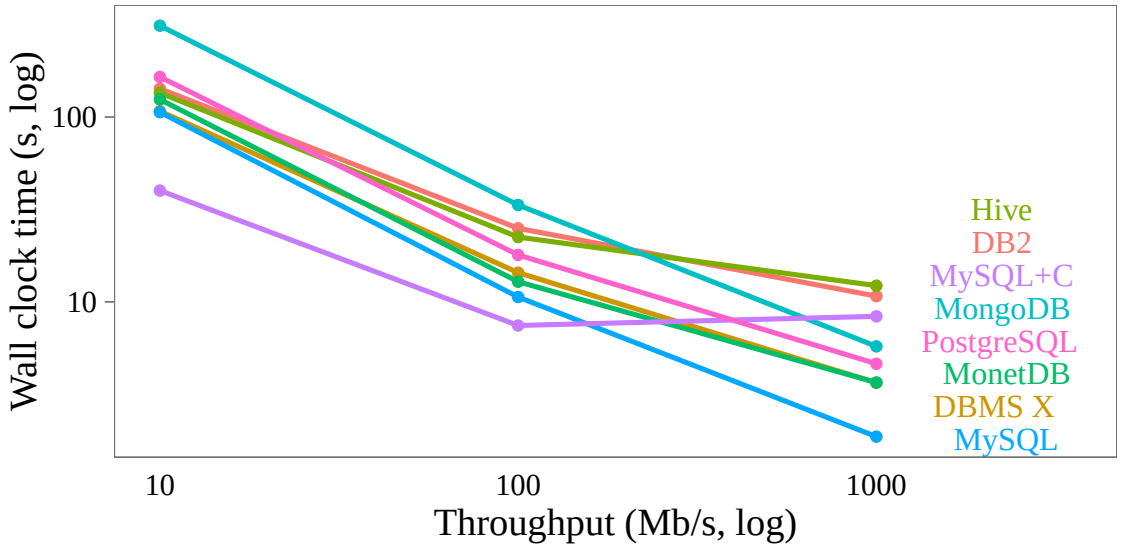



Figure 3-4: Time taken to transfer a result set with varying throughput limitations.

of the message size. High latency is particularly problematic when either the client or the server has to receive a message before it can proceed. This occurs during authentication, for example. The server sends a challenge to the client and then has to wait a full round-trip before receiving the response.

When transferring large result sets, however, such handshakes are unnecessary. While we expect a higher latency to significantly influence the time it takes to establish a connection, the transfer of a large result set should not be influenced by the latency as the server can send the entire result set without needing to wait for any confirmation. As we filter out startup costs to isolate the result set transfer, we do not expect that a higher latency will significantly influence the time it takes to transfer a result set.

In Figure 3-3, we see the influence that higher latencies have on the different protocols. We also observe that both DB2 and DBMS X perform significantly worse when the latency is increased. It is possible that they send explicit confirmation messages from the client to the server to indicate that the client is ready to receive the next batch of data. These messages are cheap with a low latency, but become very costly when the latency increases.

Contrary to our prediction, we find that the performance of all systems is heavily

influenced by a high latency. This is because, while the server and client do not explicitly send confirmation messages to each other, the underlying TCP/IP layer does send acknowledgement messages when data is received [66]. TCP packets are sent once the underlying buffer fills up, resulting in an acknowledgement message. As a result, protocols that send more data trigger more acknowledgements and suffer more from a higher latency.

**Throughput.** Reducing the throughput of a connection adds a variable cost to sending messages depending on the size of the message. Restricted throughput means sending more bytes over the socket becomes more expensive. The more we restrict the throughput, the more protocols that send a lot of data are penalized.

In Figure 3-4, we can see the influence that lower throughputs have on the different protocols. When the bandwidth is reduced, protocols that send a lot of data start performing worse than protocols that send a lower amount of data. While the PostgreSQL protocol performs well with a high throughput, it starts performing significantly worse than the other protocols with a lower throughput. Meanwhile, we also observe that when the throughput decreases compression becomes more effective. When the throughput is low, the actual data transfer is the main bottleneck and the cost of (de)compressing the data becomes less significant.

## 2.3 Result Set Serialization

In order to better understand the differences in time and transferred bytes between the different protocols, we have investigated their data serialization formats.

Table 3.2: Simple result set table.

| INT32 | VARCHAR10 |
|---:|---|
| 100,000,000 | OK |
| NULL | DPFKG |

For each of the protocols, we show a hexadecimal representation of Table 3.2 encoded with each result set format. The bytes used for the actual data are colored green, while any overhead is colored white. For clarity, leading zeroes are colored gray.

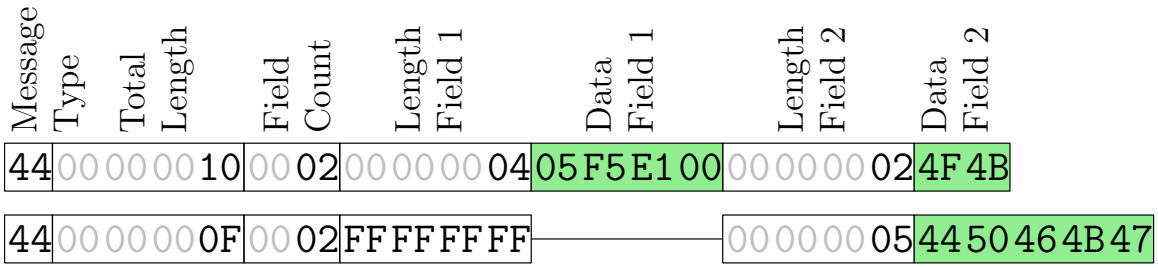



Figure 3-5: PostgreSQL result set wire format

**PostgreSQL.** Figure 3-5 shows the result set serialization of the *widely used* PostgreSQL protocol. In the PostgreSQL result set, every single row is transferred in a separate protocol message [83]. Each row includes a total length, the amount of fields, and for each field its length ($-1$ if the value is `NULL`) followed by the data. We can see that for this result set, the amount of *per-row* metadata is greater than the actual data w.r.t. the amount of bytes. Furthermore, a lot of information is repetitive and redundant. For example, the amount of fields is expected to be constant for an entire result set. Also, from the result set header that precedes those messages, the amount of rows in the result set is known, which makes the message type marker redundant. This large amount of redundant information explains why PostgreSQL's client protocol requires so many bytes to transfer the result set in the experiment shown in Table 3.1. On the other hand, the simplicity of the protocol results in low serialization and deserialization costs. This is reflected in its quick transfer time if the network connection is not a bottleneck.

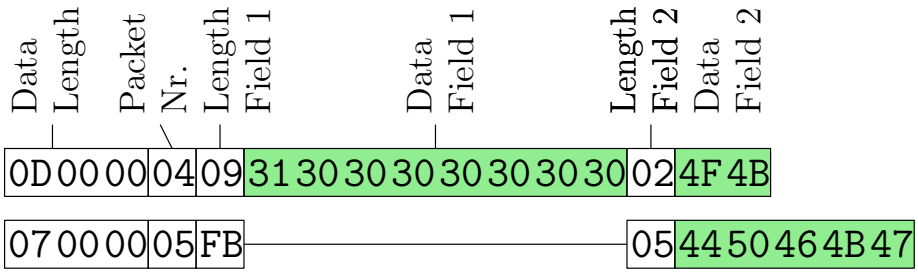



Figure 3-6: MySQL text result set wire format

**MySQL.** Figure 3-6 shows MySQL/MariaDB's protocol encoding of the sample result set. The protocol uses binary encoding for metadata, and text for actual field

data. The number of fields in a row is constant and defined in the result set header. Each row starts with a three-byte data length. Then, a packet sequence number (0-256, wrapping around) is sent. This is followed by length-prefixed field data. Field lengths are encoded as variable-length integers. `NULL` values are encoded with a special field length, `0xFB`. Field data is transferred in ASCII format. The sequence number is redundant here as the underlying TCP/Unix Sockets already guarantees that packets arrive in the same order in which they were sent.

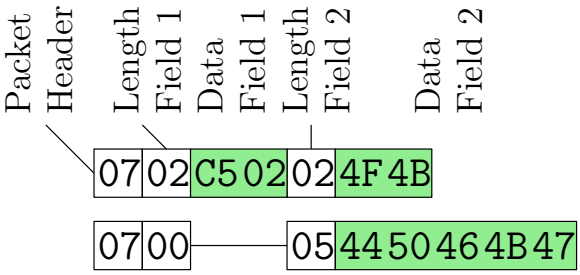



Figure 3-7: DBMS X result set wire format

**DBMS X** has a very terse protocol. However, it is much more computationally heavy than the protocol used by *PostgreSQL*. Each row is prefixed by a packet header, followed by the values. Every value is prefixed by its length in bytes. This length, however, is transferred as a variable-length integer. As a result, the length-field is only a single byte for small lengths. For `NULL` values, the length field is `0` and no actual value is transferred. Numeric values are also encoded using a custom format. On a lower layer, DBMS X uses a fixed network message length for batch transfers. This message length is configurable and according to the documentation, considerably influences performance. We have set it to the largest allowed value, which gave the best performance in our experiments.

**MonetDB.** Figure 3-8 shows MonetDB's text-based result serialization format. Here, the ASCII representations of values are transferred. This side-steps some issues with endian-ness, transfer of leading zeroes and variable-length strings. Again, every result set row is preceded by a message type. Values are delimited similar to CSV files. A newline character terminates the result row. Missing values are encoded as the string literal `NULL`. In addition (for historic reasons), the result set format includes

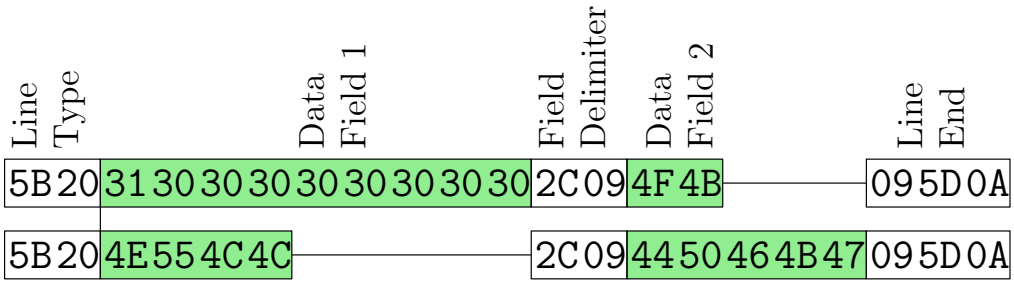



Figure 3-8: MonetDB result set wire format

formatting characters (tabs and spaces), which serve no purpose here but inflate the size of the encoded result set. While it is simple, converting the internally used binary value representations to strings and back is an expensive operation.

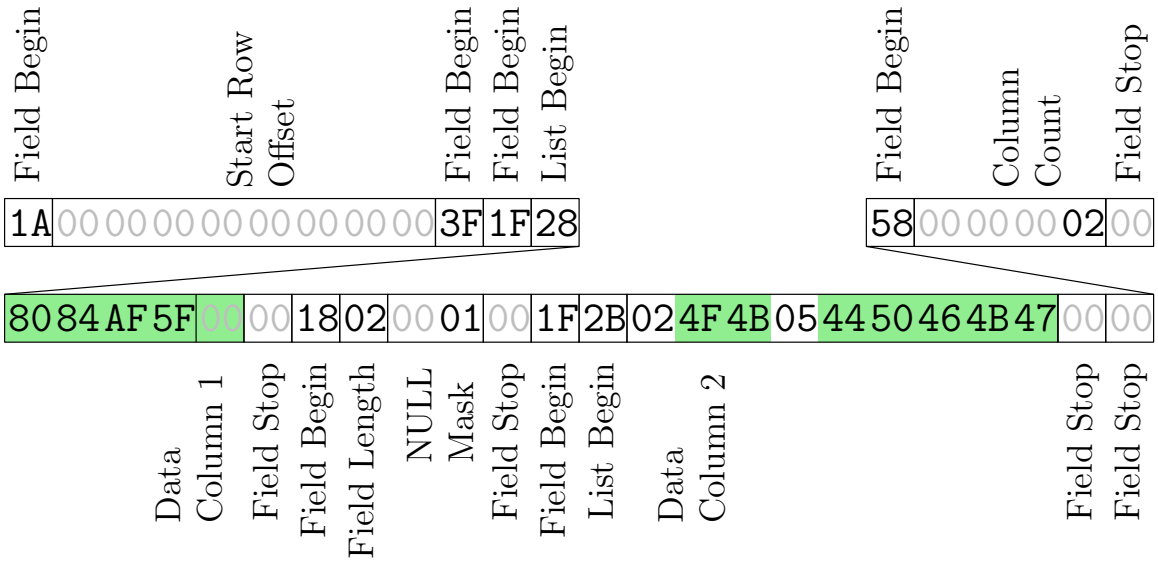



Figure 3-9: Hive result set wire format using "compact" Thrift encoding

**Hive.** Hive and Spark SQL use a Thrift-based protocol to transfer result sets [68]. Figure 3-9 shows the serialization of the example result set. From Hive version 2 onwards, a columnar result set format is used. Thrift contains a serialization method for generic complex structured messages. Due to this, serialized messages contain various meta data bytes to allow reassembly of the structured message on the client side. This is visible in the encoded result set. Field markers are encoded as a single byte if possible, the same holds for list markers which also include a length.

This result set serialization format is unnecessarily verbose. However, due to the columnar nature of the format, these overheads are not dependent on the number of rows in the result set. The only per-value overheads are the lengths of the string values and the `NULL` mask. The `NULL` mask is encoded as one byte per value, wasting a significant amount of space.

Despite the columnar result set format, Hive performs very poorly on our benchmark. This is likely due to the relatively expensive variable-length encoding of each individual value in integer columns.

# 3 Protocol Design Space

In this section, we will investigate several trade-offs that must be considered when designing a result set serialization format. The protocol design space is generally a trade-off between computation and transfer cost. If computation is not an issue, heavy-weight compression methods such as XZ [72] are able to considerably reduce the transfer cost. If transfer cost is not an issue (for example when running a client on the same machine as the database server) performing less computation at the expense of transferring more data can considerably speed up the protocol.

In the previous section, we have seen a large number of different design choices, which we will explore here. To test how each of these choices influence the performance of the serialization format, we benchmark them in isolation. We measure the wall clock time of result set (de)serialization and transfer and the size of the transferred data. We perform these benchmarks on three datasets.

- **lineitem** from the TPC-H benchmark. This table is designed to be similar to real-world data warehouse fact tables. It contains 16 columns, with the types of either `INTEGER`, `DECIMAL`, `DATE` and `VARCHAR`. This dataset contains no missing values. We use the SF10 `lineitem` table, which has 60 million rows and is 7.2GB in CSV format.

- **American Community Survey (ACS)** [10]. This dataset contains millions

of census survey responses. It consists of 274 columns, with the majority of type `INTEGER`. 16.1% of the fields contain missing values. The dataset has 9.1 million rows, totaling 7.0GB in CSV format.

- **Airline On-Time Statistics** [62]. The dataset describes commercial air traffic punctuality. The most frequent types in the 109 columns are `DECIMAL` and `VARCHAR`. 55.2% of the fields contain missing values. This dataset has 10 million rows, totaling 3.6GB in CSV format.

## 3.1 Protocol Design Choices

**Row/Column-wise.** As with storing tabular data on sequential storage media, there is also a choice between sending values belonging to a single row first versus sending values belonging to a particular column first. In the previous section, we have seen that most systems use a row-wise serialization format regardless of their internal storage layout. This is likely because predominant database APIs such as ODBC and JDBC focus heavily on row-wise access, which is simpler to support if the data is serialized in a row-wise format as well. Database clients that print results to a console do so in a row-wise fashion as well.

Yet we expect that column-major formats will have advantages when transferring large result sets, as data stored in a column-wise format compresses significantly better than data stored in a row-wise format [1]. Furthermore, popular data analysis systems such as the R environment for statistical computing [69] or the Pandas Python package [56] also internally store data in a column-major format. If data to be analysed with these or similar environments is retrieved from a modern columnar or vectorised database using a traditional row-based socket protocol, the data is first converted to row-major format and then back again. This overhead is unnecessary and can be avoided.

The problem with a pure column-major format is that an entire column is transferred before the next column is sent. If a client then wants to provide access to the data in a row-wise manner, it first has to read and cache the entire result set. For

large result sets, this can be infeasible.

Our chosen compromise between these two formats is a *vector-based protocol*, where chunks of rows are encoded in column-major format. To provide row-wise access, the client then only needs to cache the rows of a single chunk, rather than the entire result set. As the chunks are encoded in column-major order, we can still take advantage of the compression and performance gains of a columnar data representation. This trade-off is similar to the one taken in vector-based systems such as VectorWise [9].

Table 3.3: Transferring each of the datasets with different chunk sizes.

| | Chunksize | Rows | Time | Size (GB) | C. Ratio |
|---|---|---|---|---|---|
| Lineitem | 2KB | $1.4\times10^1$ | 55.9 | 6.56 | 1.38 |
| | 10KB | $7.1\times10^1$ | 15.2 | 5.92 | 1.80 |
| | 100KB | $7.1\times10^2$ | 10.9 | 5.81 | 2.12 |
| | 1MB | $7.1\times10^3$ | **10.0** | **5.80** | 2.25 |
| | 10MB | $7.1\times10^4$ | 10.9 | **5.80** | **2.26** |
| | 100MB | $7.1\times10^5$ | 13.3 | 6.15 | 2.23 |
| ACS | 2KB | $1.0\times10^0$ | 281.1 | 11.36 | 2.06 |
| | 10KB | $8.0\times10^0$ | 46.7 | 9.72 | 3.18 |
| | 100KB | $8.5\times10^1$ | 16.2 | 9.50 | 3.68 |
| | 1MB | $8.5\times10^2$ | **11.9** | **9.49** | 3.81 |
| | 10MB | $8.5\times10^3$ | 15.3 | 9.50 | **3.86** |
| | 100MB | $8.5\times10^4$ | 17.9 | 10.05 | 3.84 |
| Ontime | 2KB | $1.0\times10^0$ | 162.9 | 8.70 | 2.13 |
| | 10KB | $8.0\times10^0$ | 27.3 | 4.10 | 4.15 |
| | 100KB | $8.5\times10^1$ | 7.6 | 3.47 | 8.15 |
| | 1MB | $8.6\times10^2$ | 6.9 | 3.42 | 9.80 |
| | 10MB | $8.6\times10^3$ | **6.2** | **3.42** | 10.24 |
| | 100MB | $8.6\times10^4$ | 11.9 | 3.60 | **10.84** |

**Chunk Size.** When sending data in chunks, we have to determine how large these chunks will be. Using a larger chunk size means both the server and the client need to allocate more memory in their buffer, hence we prefer smaller chunk sizes. However, if we make the chunks too small, we do not gain any of the benefits of a columnar protocol as only a small number of rows can fit within a chunk.

To determine the effect that larger chunk sizes have on the wall clock time and compression ratio we experimented with various different chunk sizes using the three different datasets. We sent all the data from each dataset with both the

uncompressed columnar protocol, and the columnar protocol compressed with the lightweight compression method *Snappy* [46]. We varied the chunk size between 2KB and 100MB. The minimum of 2KB was chosen so a single row of each dataset can fit within a chunk. We measure the total amount of bytes that were transferred, the wall clock time required and the obtained compression ratio.

In Table 3.3 we can see the results of this experiment. For each dataset, the protocol performs poorly when the chunk size is very small. In the worst case, only a single row can fit within each chunk. In this scenario, our protocol is similar to a row-based protocol. We also observe that the protocol has to transfer more data and obtains a poor compression ratio when the chunk size is low.

However, we can see that both the performance and the compression ratio converge relatively quickly. For all three datasets, the performance is optimal when the chunk size is around 1MB. This means that the client does not need a large amount of memory to get good performance with a vector-based serialization format.

**Data Compression.** If network throughput is limited, compressing the data that is sent can greatly improve performance. However, data compression comes at a cost. There are various generic, data-agnostic compression utilities that each make different trade-offs in terms of the (de)compression costs versus the achieved compression ratio. The lightweight compression tools Snappy [46] and LZ4 [18] focus on fast compression and sacrifice compression ratio. XZ [72], on the other hand, compresses data very slowly but achieves very tight compression. GZIP [31] obtains a balance between the two, achieving a good compression ratio while not being extremely slow.

To test each of these compression methods, we have generated both a column-major and a row-major protocol message containing the data of one million rows of the `lineitem` table. All the data is stored in binary format, with dates stored as four-byte integers resembling the amount of days since 0 AD and strings stored as null delimited values.

Table 3.4 shows the compression ratios on both the row-wise and the column-wise binary files. We can see that even when using generic, data-agnostic compression methods the column-wise files always compress significantly better. As expected,

Table 3.4: Compression ratio of row/column-wise binary files

| Method | | Size (MB) | C. Ratio |
|---|---|---|---|
| LZ4 | Column | **50.0** | **2.10** |
| | Row | 57.0 | 1.85 |
| Snappy | Column | **47.8** | **2.20** |
| | Row | 54.8 | 1.92 |
| GZIP | Column | **32.4** | **3.24** |
| | Row | 38.1 | 2.76 |
| XZ | Column | **23.7** | **4.44** |
| | Row | 28.1 | 3.74 |

the heavyweight compression tools achieve a better compression ratio than their lightweight counterparts.

However, compression ratio does not tell the whole story when it comes to stream compression. There is a trade-off between heavier compression methods that take longer to compress the data while transferring fewer bytes and more lightweight compression methods that have a worse compression ratio but (de)compress data significantly faster. The best compression method depends on how expensive it is to transfer bytes; on a fast network connection a lightweight compression method performs better because transferring additional bytes is cheap. On a slower network connection, however, spending additional time on computation to obtain a better compression ratio is more worthwhile.

To determine which compression method performs better at which network speed, we have run a benchmark where we transfer the SF10 `lineitem` table over a network connection with different throughput limitations.

Table 3.5: Compression effectiveness vs. cost

| | | Timings (s) | | | | |
|---|---|---|---|---|---|---|
| | Comp | $T_{local}$ | $T_{1000}$ | $T_{100}$ | $T_{10}$ | Size (MB) |
| Lineitem | None | **1.5** | 10.4 | 84.8 | 848 | 1012 |
| | Snappy | 3.3 | **3.8** | **37.3** | 373 | 447 |
| | LZ4 | 4.5 | 4.9 | 38.4 | 383 | 456 |
| | GZIP | 59.8 | 60.4 | 59.6 | **226** | 272 |
| | XZ | 695 | 689 | 666 | 649 | **203** |

The results of this experiment are shown in Table 3.5. We can see that not compressing the data performs best when the server and client are located on the same machine. Lightweight compression becomes worthwhile when the server and client are using a gigabit or worse connection (1 Gbit/s). In this scenario, the uncompressed protocol still performs better than heavyweight compression techniques. It is only when we move to a very slow network connection (10Mbit/s) that heavier compression performs better than lightweight compression. Even in this case, however, the very heavy XZ still performs poorly because it takes too long to compress/decompress the data.

The results of this experiment indicate that the best compression method depends entirely on the connection speed between the server and the client. Forcing manual configuration for different setups is a possibility but is cumbersome for the user. Instead, we choose to use a simple heuristic for determining which compression method to use. If the server and client reside on the same machine, we do not use any compression. Otherwise, we use lightweight compression, as this performs the best in most realistic network use cases where the user has either a LAN connection or a reasonably high speed network connection to the server.

**Column-Specific Compression.** Besides generic compression methods, it is also possible to compress individual columns. For example, run-length encoding or delta encoding could be used on numeric columns. The database also could have statistics on a column which would allow for additional optimizations in column compression. For example, with min/max indexes we could select a bit packing length for a specific column without having to scan it.

Using these specialized compression algorithms we could achieve a higher compression ratio at a lower cost than when using data-agnostic compression algorithms. Integer values in particular can be compressed at a very high speed using vectorized binpacking or PFOR [51] compression algorithms.

To investigate the performance of these specialized integer compression algorithms, we have performed an experiment in which we transfer *only the integer columns* of the three different datasets. The reason we transfer only the integer columns is because

these compression methods are specifically designed to compress integers, and we want to isolate their effectiveness on these column types. The `lineitem` table has 8 integer columns, the ACS dataset has 265 integer columns and the ontime dataset has 17 integer columns.

For the experiment, we perform a projection of only the integer columns in these datasets and transfer the result of the projection to the client. We test both the specialized compression methods PFOR and binpacking, and the generic compression method Snappy. The PFOR and binpacking compression methods compress the columns individually, whereas Snappy compresses the entire message at once. We test each of these configurations on different network configurations, and measure the wall clock time and bytes transferred over the socket.

Table 3.6: Cost for retrieving the *int* columns using different compression methods.

| | | Timings (s) | | | |
|---|---|---|---|---|---|
| | System | $T_{Local}$ | $T_{LAN}$ | $T_{WAN}$ | Size (MB) |
| Lineitem | None | **5.3** | 15.7 | 159.0 | 1844.2 |
| | Binpack | 6.0 | 8.0 | 82.0 | 944.1 |
| | PFOR | 5.7 | 8.1 | 82.1 | 948.0 |
| | Snappy | 6.8 | 12.3 | 103.9 | 1204.9 |
| | Binpack+Sy | 5.8 | **7.5** | **76.4** | **882.0** |
| | PFOR+Sy | 5.7 | **7.5** | 77.5 | 885.9 |
| ACS | None | 15.2 | 78.6 | 800.6 | 9244.8 |
| | Binpack | 120.5 | 133.9 | 421.2 | 4288.2 |
| | PFOR | 166.8 | 170.1 | 300.9 | 2703.4 |
| | Snappy | **20.5** | **22.8** | 204.5 | 2434.8 |
| | Binpack+Sy | 152.6 | 160.9 | 190.0 | 1694.6 |
| | PFOR+Sy | 165.8 | 168.4 | **185.4** | **1203.2** |
| Ontime | None | **1.3** | 5.8 | 54.4 | 649.1 |
| | Binpack | 1.4 | 6.2 | 44.4 | 529.3 |
| | PFOR | 1.6 | 5.8 | 44.4 | 528.6 |
| | Snappy | 1.4 | **1.4** | **3.2** | **39.0** |
| | Binpack+Sy | 1.8 | 1.9 | 5.7 | 67.7 |
| | PFOR+Sy | 1.8 | 1.9 | 5.9 | 70.5 |

In Table 3.6, the results of this experiment are shown. For the lineitem table, we see that both PFOR and binpacking achieve a higher compression ratio than Snappy at a lower performance cost. As a result, these specialized compression algorithms

perform better than Snappy in all scenarios. Combining the specialized compression methods with Snappy allows us to achieve an even higher compression ratio. We still note that not compressing performs better in the localhost scenario, however.

When transferring the ACS dataset the column-specific compression methods perform significantly worse than Snappy. Because a large amount of integer columns are being transferred (265 columns) each chunk we transfer contains relatively few rows. As a result, the column-specific compression methods are called many times on small chunks of data, which causes poor performance. Snappy is unaffected by this because it does not operate on individual columns, but compresses the entire message instead.

We observe that the PFOR compression algorithm performs significantly better than binpacking on the ACS data. This is because binpacking only achieves a good compression ratio on data with many small values, whereas PFOR can efficiently compress columns with many large numbers as long as the values are close together.

Both specialized compression algorithms perform very poorly on the ontime dataset. This dataset has both large values, and a large difference between the minimum and maximum values. However, Snappy does obtain a very good compression ratio. This is because values that are close together are similar, making the dataset very compressible.

Overall, we can see that the specialized compression algorithms we have tested can perform better than Snappy on certain datasets. However, they do not perform well on all data distributions and they require each message to contain many rows to be effective. As a result, we have chosen not to use column-specific compression algorithms. As future work it would be possible to increase protocol performance by choosing to use these specialized compression algorithms based on database statistics.

**Data Serialization.** The sequential nature of the TCP sockets requires an organized method to write and read data from them. Options include custom text/binary serializations or generic serialization libraries such as Protocol Buffers [34] or Thrift [68]. We can expect that the closer the serialized format is to the native data storage layout, the less the computational overhead required for their (de)serialization.

To determine the performance impact that generic serialization libraries have when serializing large packages, we perform an experiment in which we transfer the `lineitem` table using both a custom serialization format and protocol buffers. For both scenarios, we test an uncompressed protocol and a protocol compressed with Snappy.

Table 3.7: Cost for transferring data using a custom serialization format vs protocol buffers.

| | | Timings (s) | | | |
|---|---|---|---|---|---|
| | System | $T_{Local}$ | $T_{LAN}$ | $T_{WAN}$ | Size (MB) |
| Lineitem | Custom | **10.3** | 64.1 | 498.9 | 5943.3 |
| | Custom+C | 18.3 | **25.4** | 221.4 | 2637.4 |
| | Protobuf | 33.1 | 45.5 | 391.6 | 4656.1 |
| | Protobuf+C | 35.7 | 47.3 | **195.2** | **2315.9** |

In Table 3.7, the results of this experiment are shown. We can see that our custom result set serialization format performs significantly better than protobuf serialization. This is because protobuf operates as a generic protocol and does not consider the context of the client-server communication. Protobuf will, for example, perform unnecessary endianness conversions on both the server- and client- side because it does not know that the server and client use the same endianness. As a result of these unnecessary operations, the (un)packing of protobuf messages is very expensive.

We do see that protobuf messages are smaller than our custom format. This is because protobuf messages store integers as varints, saving space for small integer values. However, protocol buffers achieve a very small compression ratio at a very high cost compared to actual compression algorithms. As a result of these high serialization costs, we have chosen to use a custom serialization format.

**String handling.** Character strings are one of the more difficult cases for serialization. There are three main options for character transfer.

- Null-Termination, where every string is suffixed with a `0` byte to indicate the end of the string.
- Length-Prefixing, where every string is prefixed with its length.

- Fixed Width, where every string has a fixed width as described in its SQL type.

Each of these approaches has a number of advantages and disadvantages. Strings encoded with length-prefixing need additional space for the length. This can drastically increase the size of the protocol message, especially when there are many small strings. This effect can be mitigated by using variable-length integers. This way, small strings only require a single byte for their length. However, variable integers introduce some additional computation overhead, increasing (de)serialization time.

Null-Termination only requires a single byte of padding for each string, however, the byte is always the same value and is therefore very compressible. The disadvantage of null-termination is that the client has to scan the entire string to find out where the next string is. With length-prefixing, the client can read the length and jump that many bytes ahead.

Fixed-Width has the advantage that there is no unnecessary padding if each string has the same size. However, in the case of `VARCHARs`, this is not guaranteed. If there are a small amount of long strings and a large amount of short (or NULL) strings, fixed-width encoding can introduce a significant amount of unnecessary padding.

To determine how each of these string representations perform, we have tested each of these approaches by transferring different string columns of the `lineitem` table. For each experiment, we transfer 60 million rows of the specified column with both the uncompressed protocol and the protocol compressed with Snappy.

Table 3.8: Transferring the `l_returnflag` column of the SF10 lineitem table.

| Type | Time | Time+C | Size(MB) | C.Ratio |
|---|---|---|---|---|
| Varint Prefix | 3.94 | 3.99 | 114.54 | 3.37 |
| Null-Terminated | 3.95 | 3.91 | 114.54 | 3.37 |
| `VARCHAR(1)` | **3.68** | **3.76** | **57.34** | **2.84** |

In Table 3.8, the result of transferring only the single-character column `l_returnflag` is shown. As expected, we can see that a fixed-width representation performs extremely well while transferring a small string column. Both the length-prefix and null-terminated approaches use an additional byte per string, causing them to transfer twice the amount of bytes.

Table 3.9: Transferring the `l_comment` column of the SF10 lineitem table.

| Type | Time | Time+C | Size(GB) | C.Ratio |
|---|---|---|---|---|
| Null-Terminated | **4.12** | **6.09** | **1.53** | 2.44 |
| Varint Prefix | 4.24 | 6.63 | **1.53** | 2.27 |
| `VARCHAR(44)` | 4.15 | 7.66 | 2.46 | 3.12 |
| `VARCHAR(100)` | 5.07 | 10.13 | 5.59 | 5.69 |
| `VARCHAR(1000)` | 16.71 | 26.30 | 55.90 | 15.32 |
| `VARCHAR(10000)` | 171.55 | 216.23 | 559.01 | **20.19** |

In Table 3.9, the result of transferring the longer column `l_comment` is shown. This column has a maximum string length of `44`. We can see that all the approaches have comparable performance when transferring this column. However, the fixed-width approach transfers a significantly higher number of bytes. This is because many of the strings are not exactly 44 characters long, and hence have to be padded. As a result of more data being transferred, the compression is also more expensive.

To illustrate the effect that this unnecessary padding can have on performance in the worst case, we have repeated this experiment with different `VARCHAR` type widths. We note that as we increase the width of the `VARCHAR` type, the amount of data that the fixed-width approach has to transfer drastically increases. While the compressibility does significantly improve with the amount of padding, this does not sufficiently offset the increased size.

The results of these experiments indicate that the fixed-width representation is well suited for transferring narrow string columns, but has a very poor worst-case scenario when dealing with wider string columns. For this reason, we have chosen to conservatively use the fixed-width representation only when transferring columns of type `VARCHAR(1)`. Even when dealing with `VARCHAR` columns of size two, the fixed-width representation can lead to a large increase in transferred data when many of the strings are empty. For larger strings, we use the null-termination method because of its better compressibility.

# 4 Implementation & Results

In the previous section, we have investigated several trade-offs that must be considered when designing a protocol. In this section we will describe the design of our own protocol, and its implementation in PostgreSQL and MonetDB. Afterwards, we will provide an extensive evaluation comparing the performance of our protocol with the state of the art client protocols when transferring large amounts of real-world data.

## 4.1 MonetDB Implementation

Figure 3-10 shows the serialization of the data from Table 3.2 with our proposed protocol in MonetDB. The query result is serialized to column-major chunks. Each chunk is prefixed by the amount of rows in that particular chunk. After the row count, the columns of the result set follow in the same order as they appear in the result set header. Columns with fixed-length types, such as four-byte integers, do not have any additional stored before them. Columns with variable-length types, such as `VARCHAR` columns, are prefixed with the total length of the column in bytes. Using this length, the client can access the next column in the result set without having to scan through the variable-length column. This allows the client to efficiently provide row-wise access to the data.

Missing values are encoded as a special value within the domain of the type being transferred. For example, the value $2^{-31}$ is used to represent the `NULL` value for four-byte integers. This approach is used internally by MonetDB to store missing values, and is efficient when there are no or few missing values to be transferred.

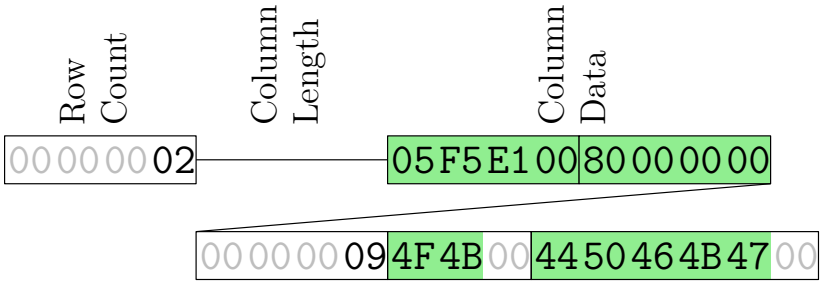



Figure 3-10: Proposed result set wire format – MonetDB

The maximum size of the chunks is specified in bytes. The maximum chunk size is set by the client during authentication. The advantage to this approach is that the size of the chunks does not depend on the width of the rows. This way, the client only needs to allocate a single buffer to hold the result set messages. Chunks will always fit within that buffer outside of the edge case when there are extremely wide rows. The client can then read an entire chunk into that buffer, and directly access the data stored without needing to unnecessarily convert and/or copy the data.

When the server sends a result set, the server chooses the amount of rows to send such that the chunk does not exceed the maximum size. If a single row exceeds this limit, the server will send a message to the client indicating that it needs to increase the size of its buffer so a single row can fit within it. After choosing the amount of rows that fit within a chunk, the server copies the result into a local buffer in column-wise order. As MonetDB stores the data in column-wise order, the data of each of the columns is copied sequentially into the buffer. If column-specific compression is enabled for a specific column, the data is compressed directly into the buffer instead of being copied. After the buffer is filled, the server sends the chunk to the client. If chunk-wise compression is enabled, the entire chunk is compressed before being transferred.

Note that choosing the amount of rows to send is typically a constant operation. Because we know the maximum size of each row for most column types, we can compute how many rows can fit within a single chunk without having to scan the data. However, if there are BLOB or CLOB columns every row can have an arbitrary size. In this case, we perform a scan over the elements of these columns to determine how many rows we can fit in each chunk. In these cases, the amount of rows per chunk can vary on a per-chunk basis.

## 4.2 PostgreSQL Implementation

Figure 3-11 shows the serialization of the data from Table 3.2 with our proposed protocol in PostgreSQL. Like the proposed protocol in MonetDB, the result is serialized to column-major chunks and prefixed by the amount of rows in that chunk. However,

missing values are encoded differently. Instead of a special value within the domain, each column is prefixed with a bitmask that indicates for each value whether or not it is missing. When a missing value is present, the bit for that particular row is set to 1 and no data value is transferred for that row. Because of this bitmask, even columns with fixed-width types now have a variable length. As such, every column is now prefixed with its length to allow the client to skip past columns without scanning the data or the bitmask.

As we store the bitmask per column, we can leave out the bitmask for columns that do not have any missing values. When a column is marked with the `NOT NULL` flag or database statistics indicate that a column does not contain missing values, we do not send a `NULL` mask. In the result set header, we notify the client which columns have a `NULL` mask and which do not. This allows us to avoid unnecessary overhead for columns that are known to not contain any missing values.

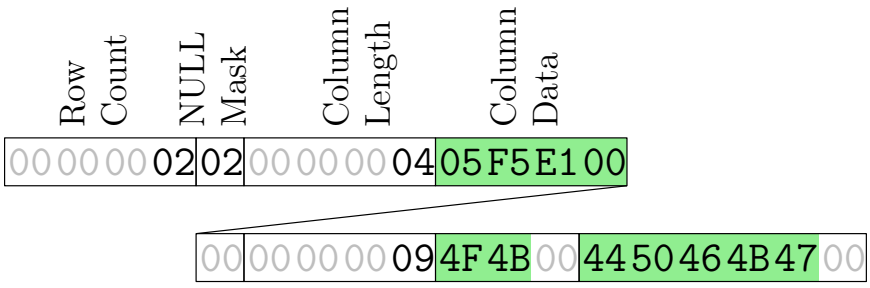



Figure 3-11: Proposed result set wire format – PostgreSQL

As PostgreSQL stores data in a row-major format, converting it to a columnar result set format provides some additional challenges. Because of the null mask, we do not know the exact size of the columns in advance, even if they have fixed-length types. To avoid wasting a lot of space when there are many missing values, we first copy the data of each column to a temporary buffer as we iterate over the rows. Once the buffer fills up, we copy the data for each column to the stream buffer and transfer it to the client.

Another potential performance issue is the access pattern of copying data in a row-major format to a column-major format. However, the cost of this random access

pattern is mitigated because the chunks are small and generally fit in the L3 cache of a CPU.

## 4.3 Evaluation

To determine how well our protocol performs in the real world, we evaluate it against the state of the art client protocols on several real world data sets.

All the experiments are performed on a Linux VM running Ubuntu 16.04. The VM has 16GB of main memory, and 8 CPU cores available. Both the database and the client run inside the same VM. The `netem` utility is used to limit the network for the slower network speed tests. The VM image, datasets and benchmark scripts are available online[1].

We perform this analysis on the `lineitem`, `acs` and `ontime` data sets described in Section 3. To present a realistic view of how our protocol performs with various network limitations, we test each dataset in three different scenarios.

- **Local.** The server and client reside on the same machine, there are no network restrictions.
- **LAN Connection.** The server and client are connected using a gigabit ethernet connection with 1000 Mb/s throughput and 0.3ms latency.
- **WAN Connection.** The server and client are connected through an internet connection, the network is restricted by 100 Mbit/s throughput and 25ms latency.

We measure all the systems described in Section 2. In addition, we measure the implementation of our protocol in MonetDB (labeled as *MonetDB++*) and our protocol in PostgreSQL (labeled as *PostgreSQL++*). As a baseline, we include the measurement of how long it takes to transfer the same amount of data in CSV format using netcat with three different compression schemes: (1) no compression, (2) compressed with Snappy, (3) compressed with GZIP. We perform this experiment using the ODBC driver of each of the respective database systems, and isolate the

[1] `https://github.com/Mytherin/Protocol-Benchmarks`

wall clock time it takes to perform result set (de)serialization and data transfer using the methods described in Section 2.1. The experiments have a timeout of 1 hour.

Table 3.10: Results of transferring the SF10 lineitem table for different network configurations.

| | | Timings (s) | | | |
|---|---|---|---|---|---|
| | System | $T_{Local}$ | $T_{LAN}$ | $T_{WAN}$ | Size |
| Lineitem | (Netcat) | (9.8) | (62.0) | (696.5) | (7.21) |
| | (Netcat+Sy) | (32.3) | (32.2) | (325.2) | (3.55) |
| | (Netcat+GZ) | (405.4) | (425.1) | (405.0) | (2.16) |
| | *MonetDB++* | **10.6** | 50.3 | 510.8 | 5.80 |
| | *MonetDB++C* | 15.5 | **19.9** | **200.6** | **2.27** |
| | *Postgres++* | 39.3 | 46.1 | 518.8 | 5.36 |
| | *Postgres++C* | 42.4 | 43.8 | 229.5 | 2.53 |
| | MySQL | 98.8 | 108.9 | 662.8 | 7.44 |
| | MySQL+C | 380.3 | 379.4 | 367.4 | 2.85 |
| | PostgreSQL | 205.8 | 301.1 | 2108.8 | 10.4 |
| | DB2 | 166.9 | 598.4 | **T** | 7.32 |
| | DBMS X | 219.9 | 282.3 | **T** | 6.35 |
| | Hive | 657.1 | 948.5 | **T** | 8.69 |
| | MonetDB | 222.4 | 256.1 | 1381.5 | 8.97 |

In Table 3.10, the results of the experiment for the `lineitem` table are shown. The timings for the different network configurations are given in seconds, and the size of the transferred data is given in gigabyte (GB).

**Lineitem.** For the `lineitem` table, we observe that our uncompressed protocol performs best in the localhost scenario, and our compressed protocol performs the best in the LAN and WAN scenarios. We note that the implementation in MonetDB performs better than the implementation in PostgreSQL. This is because converting from a row-based representation to a column-based representation requires an extra copy of all the data, leading to additional costs.

We note that DBMS X, despite its very terse data representation, still transfers significantly more data than our columnar protocol on this dataset. This is because it transfers row headers in addition to the data. Our columnar representation transfers less data because it does not transfer any per-row headers. We avoid the `NULL` mask overhead in PostgreSQL++ by not transferring a `NULL` mask for columns that are

marked as `NOT NULL`, which are all the columns in the `lineitem` table. MonetDB++ transfers missing values as special values, which incurs no additional overhead when missing values do not occur.

We also see that the timings for MySQL with compression do not change significantly when network limitations are introduced. This is because the compression of the data is interleaved with the sending of the data. As MySQL uses a very heavy compression method, the time spend compressing the data dominates the data transfer time, even with a 100Mb/s throughput limitation. However, even though MySQL uses a much heavier compression algorithm than our protocol, our compressed protocol transfers less data. This is because the columnar format that we use compresses better than the row-based format used by MySQL.

The same effect can be seen for other databases when comparing the timings of the localhost scenario with the timings of the LAN scenario. The performance of our uncompressed protocol degrades significantly when network limitations are introduced because it is bound by the network speed. The other protocols transfer data interleaved with expensive result set (de)serialization, which leads to them degrading less when minor network limitations are introduced.

The major exception to this are DBMS X and DB2. They degrade significantly when even more network limitations are introduced. This is because they both have explicit confirmation messages. DB2, especially, degrades heavily with a worse network connection.

**ACS Data.** When transferring the ACS data, we again see that our uncompressed protocol performs best in the localhost scenario and the compressed protocol performs best with network limitations.

Table 3.11: Results of transferring the ACS table for different network configurations.

| | | Timings (s) | | | |
|---|---|---|---|---|---|
| | System | $T_{Local}$ | $T_{LAN}$ | $T_{WAN}$ | Size |
| ACS | (Netcat) | (7.62) | (46.2) | (519.1) | (5.38) |
| | (Netcat+Sy) | (21.2) | (22.7) | (213.7) | (2.23) |
| | (Netcat+GZ) | (370.7) | (376.3) | (372.0) | (1.23) |
| | *MonetDB++* | **11.8** | 82.7 | 837.0 | 9.49 |
| | *MonetDB++C* | 22.0 | **22.4** | 219.0 | 2.49 |
| | *PostgreSQL++* | 43.2 | 72.0 | 787.9 | 8.24 |
| | *PostgreSQL++C* | 70.6 | 72.0 | **192.2** | 2.17 |
| | MySQL | 334.9 | 321.1 | 507.6 | 5.78 |
| | MySQL+C | 601.3 | 580.4 | 536.0 | **1.48** |
| | PostgreSQL | 277.8 | 265.1 | 1455.0 | 12.5 |
| | DB2 | 252.6 | 724.5 | **T** | 10.3 |
| | DBMS X | 339.8 | 538.1 | **T** | 6.06 |
| | Hive | 692.3 | 723.9 | 2239.2 | 9.70 |
| | MonetDB | 446.5 | 451.8 | 961.4 | 9.63 |

We can see that MySQL's text protocol is more efficient than it was when transferring the `lineitem` dataset. MySQL transfers less data than our binary protocol. In the ACS dataset, the weight columns are four-byte integers, but the actual values are rather small, typically less than 100. This favors a text representation of integers, where a number smaller than 10 only requires two bytes to encode (one byte for the length field and one for the text character).

We note that PostgreSQL performs particularly poorly on this dataset. This is because PostgreSQL' result set includes a fixed four-byte length for each field. As this dataset contains mostly integer columns, and integer columns are only four bytes wide, this approach almost doubles the size of the dataset. As a result, PostgreSQL' transfers a very large amount of bytes for this dataset.

Comparing the two new protocols, MonetDB++ and PostgreSQL++, we observe that because ACS contains a large number of `NULL` values, PostgreSQL++ transfers less data overall and thus performs better in the WAN scenario.

**Ontime Data.** As over half the values in this data set are missing, the bitmask approach of storing missing values stores the data in this result set very efficiently. As a result, we see that the PostgreSQL++ protocol transfers significantly less data

Table 3.12: Results of transferring the ontime table for different network configurations.

| | | Timings (s) | | | |
|---|---|---|---|---|---|
| | System | $T_{Local}$ | $T_{LAN}$ | $T_{WAN}$ | Size |
| Ontime | (Netcat) | (4.24) | (28.0) | (310.9) | (3.24) |
| | (Netcat+Sy) | (6.16) | (6.74) | (37.0) | (0.40) |
| | (Netcat+GZ) | (50.0) | (51.0) | (49.6) | (0.18) |
| | *MonetDB++* | **6.02** | 30.2 | 308.2 | 3.49 |
| | *MonetDB++C* | 7.16 | **7.18** | **31.3** | 0.35 |
| | *PostgreSQL++* | 13.2 | 19.2 | 213.9 | 2.24 |
| | *PostgreSQL++C* | 14.6 | 14.1 | 76.7 | 0.82 |
| | MySQL | 100.8 | 99.0 | 328.5 | 3.76 |
| | MySQL+C | 163.9 | 167.4 | 153.6 | **0.33** |
| | PostgreSQL | 111.3 | 102.8 | 836.7 | 6.49 |
| | DB2 | 113.2 | 314.1 | 3386.8 | 3.41 |
| | DBMS X | 149.9 | 281.1 | 1858.8 | 2.29 |
| | Hive | 1119.1 | 1161.3 | 2418.9 | 5.86 |
| | MonetDB | 131.6 | 135.0 | 734.7 | 6.92 |

than the MonetDB++ protocol. However, we note that the MonetDB++ protocol compresses significantly better. We speculate that this is due to the high repetitiveness of the in-column `NULL` representation which the compression method could detect as a recurring pattern and compress efficiently compared to the rather high-entropy bit patterns created by the `NULL` mask in PostgreSQL++;

The MySQL protocol achieves the best compression due to its use of GZIP. However, it still performs much worse than both MonetDB++ and PostgreSQL++ on this dataset because heavy compression still dominates execution time.

For this dataset, we also see that the current PostgreSQL protocol performs better than on the other datasets. This is because PostgreSQL saves a lot of space when transferring missing values as it only transfers a negative field length for every `NULL` value. In addition, PostgreSQL' field length indicator does not increase the result set size much when transferring large `VARCHAR` columns. However, in the WAN scenario it performs poorly because of the large amount of bytes transferred.

# 5 Summary

In this chapter, we investigated why exporting data from a database is so expensive. We took an extensive look at state of the art client protocols, and learned that they suffer from large amounts of per-row overhead and expensive (de)serialization. These issues make exporting large amounts of data very costly.

These protocols were designed for a different use case in a different era, where network layers were unable to guarantee deliver or order of packets and where OLTP use cases and row-wise data access dominated. Database query execution engines have been heavily modified or redesigned to accommodate more analytical use cases and increased data volume. Client protocols have not kept up with those developments.

To solve these issues, we analyzed the design of each of the client protocols, and noted the deficiencies that make them unsuitable for transferring large tables. We performed an in-depth analysis of all these deficiencies, and various design choices that have to be considered when designing a client protocol. Based on this analysis, we created our own client protocol and implemented it in PostgreSQL and MonetDB. We then evaluated our protocol against the state of the art protocols on various real-life data sets, and found an order of magnitude faster performance when exporting large datasets.

# CHAPTER 4

# Vectorized UDFs in Column-Stores

## 1 Introduction

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

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

result, operations performed on individual elements are very inefficient.

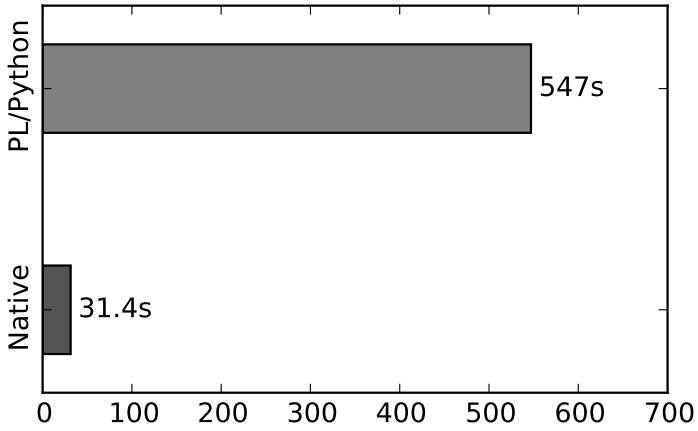



Figure 4-1: Modulo computation in Postgres.

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

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

## 1.1 Contributions

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

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

### 1.2 Outline

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

## 2 Types of User-Defined Functions

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

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

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

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

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

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

# 3 MonetDB/Python

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

## 3.1 Usage

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

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

Listing 4.1: MonetDB/Python Syntax.

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

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

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

Listing 4.2: Simple Scalar UDF.

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

## 3.2 Processing Pipeline

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

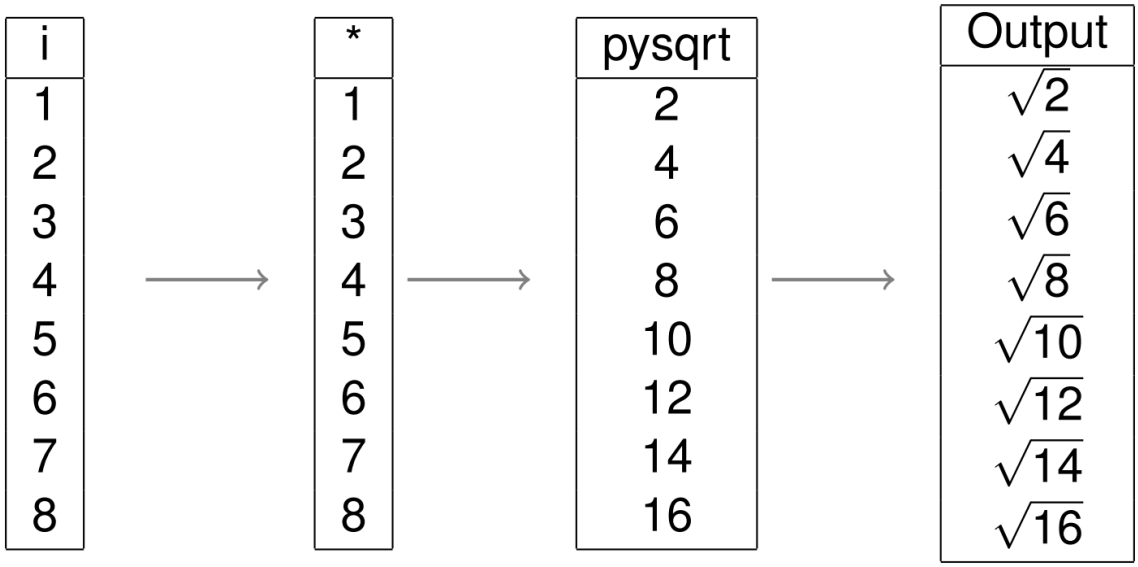



Figure 4-2: Operator Chain for Listing 4.2.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

## 3.3 Parallel Processing

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

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

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

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

Listing 4.3: Chain of SQL operators.

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

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

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

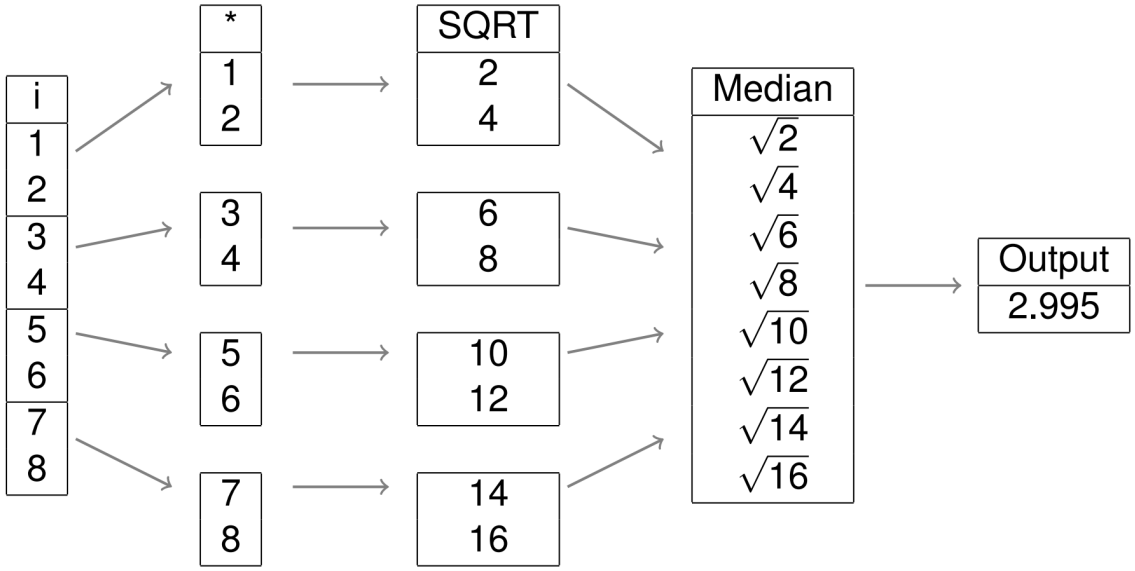



Figure 4-3: Parallel Operator Chain of Listing 4.3.

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

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

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

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

fashion.

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

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

Listing 4.4: Parallel MIN using chained operators.

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

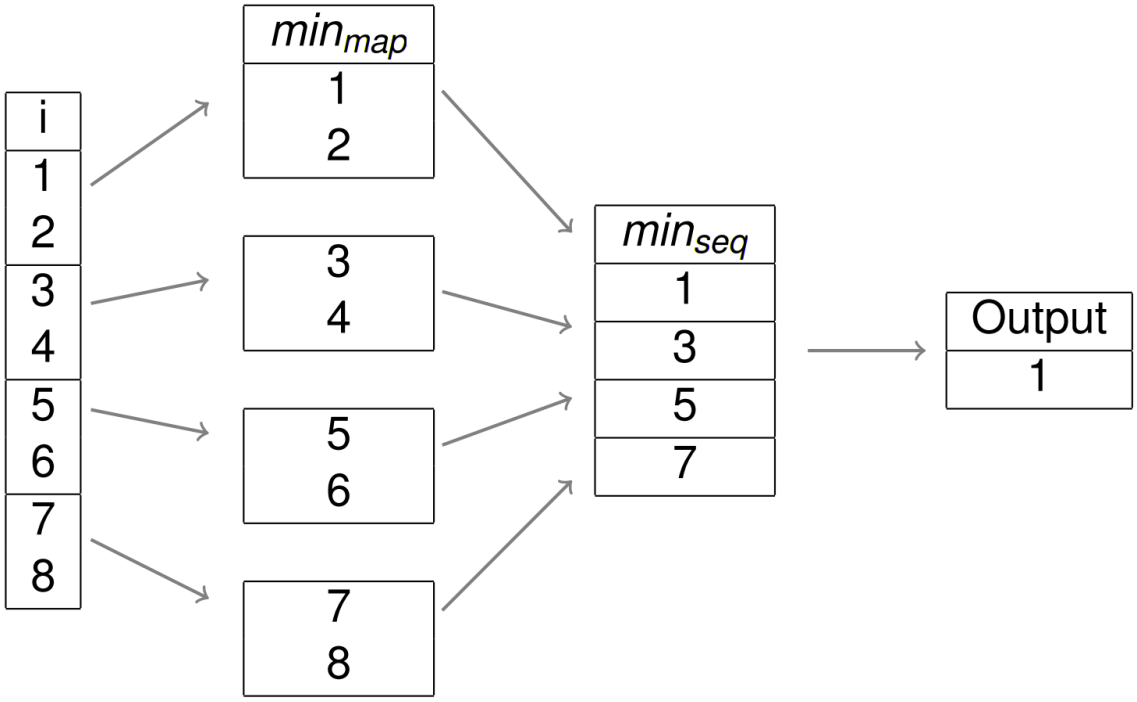



Figure 4-4: Operator Chain of Listing 4.4.

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

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

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

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

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

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

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

## 3.4 Loopback Queries

MonetDB/Python also supports loopback queries inside UDFs. Loopback queries allow users to query the database directly from within the UDF. The results of the query are converted to Python objects in a similar way as the input of the UDFs is converted. They can be can used through the `_conn` object that is passed to every UDF. Loopback queries are useful because they can bypass cardinality restrictions of the relational querying model. Listing 4.5 depicts an example of a UDF that uses a loopback query to retrieve a classifier from the database, and subsequently uses the classifier on its input data.

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

Listing 4.5: Loopback Queries

# 4 Evaluation

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

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

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

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

## 4.1 Systems Measured

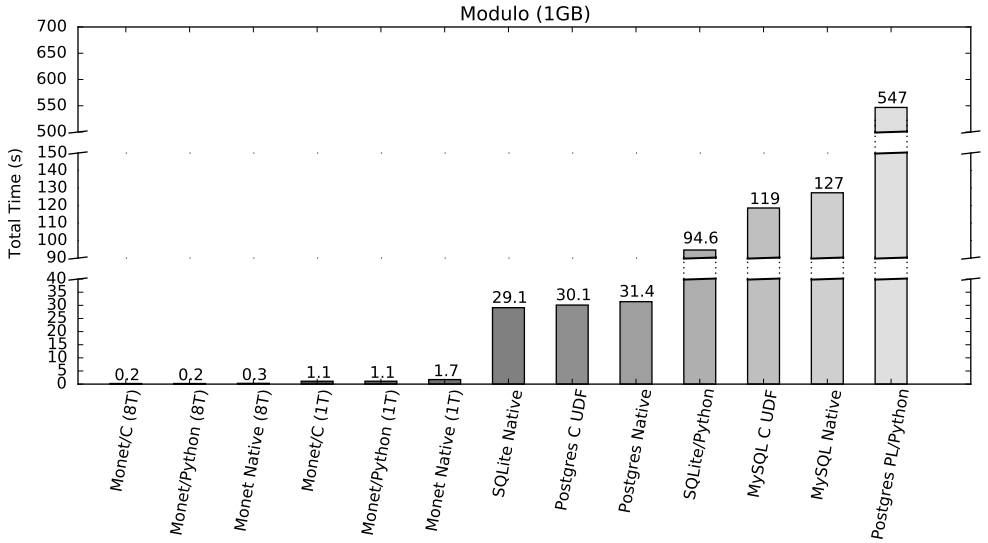



Figure 4-5: Modulo computation of 1GB of integers.

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

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

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

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

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

## 4.2 Modulo Benchmark

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

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

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

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

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

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

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

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

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

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

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

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

# 5 Related Work

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

## 5.1 Research

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

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

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

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

### In-Database Analytics

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

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

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

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

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

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

### Processing of User-Defined Functions

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

### Parallel Execution of User-Defined Functions

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

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

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

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

## 5.2 Systems

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

### Aster nCluster Database

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

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

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

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

# 6 Applicability To Other Systems

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

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

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

UDF.

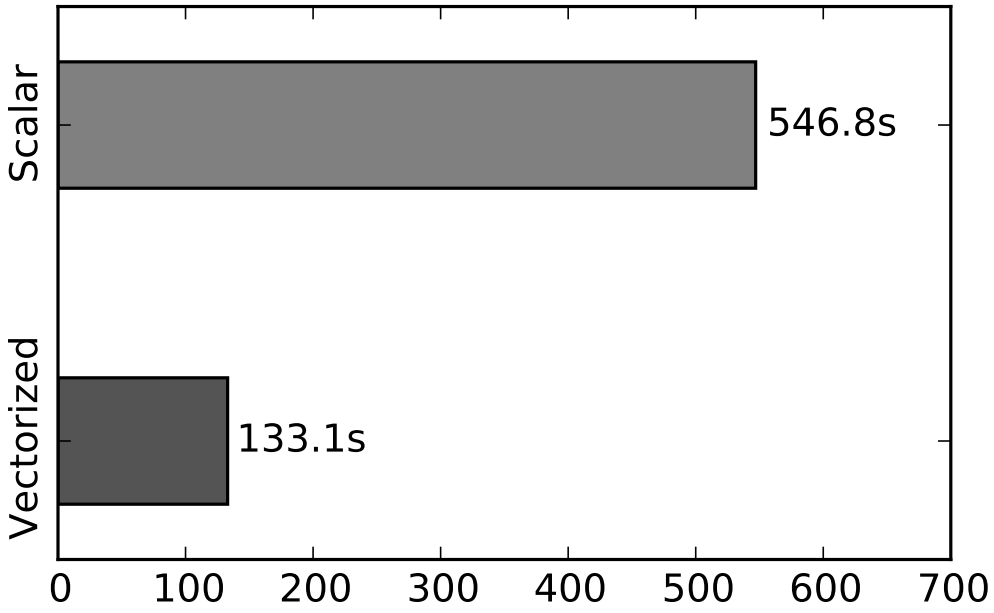



Figure 4-6: PL/Python Vectorized vs Non-Vectorized Modulo Operator.

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

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

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

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

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

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

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

# 7 Development Workflow: devUDF

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

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

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

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

Table 4.1: Most Popular Development Environments.

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

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

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

## 7.1 The devUDF Plugin

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

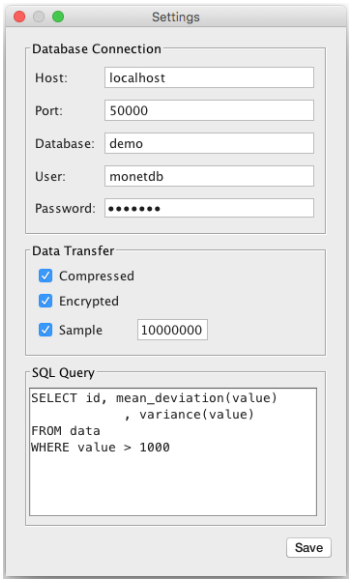



Figure 4-7: Settings.

## 7.2 Usage

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

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

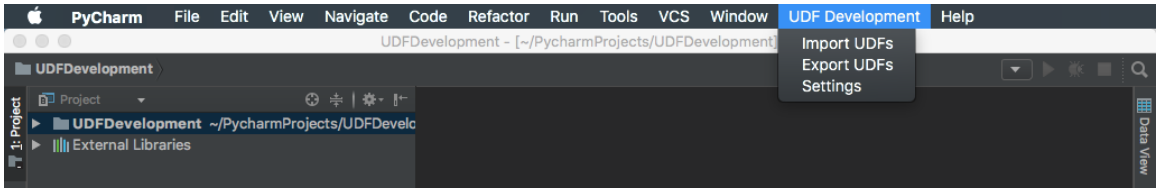
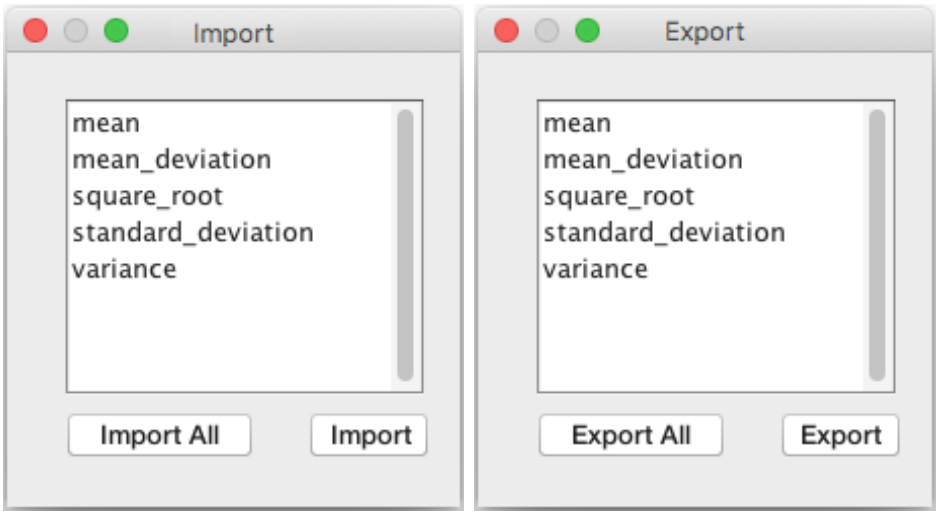



Figure 4-8: PyCharm Main Menu.




(a) Import (b) Export.

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

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

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

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

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

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

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

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

## 7.3 Implementation

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

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

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

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

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

Listing 4.6: MonetDB UDF example.

# 8 Summary

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

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

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

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

# CHAPTER 5

# In-Database Workflows

## 1 Introduction

In Chapter 4, we described the inner workings of the MonetDB/Python UDFs. By utilizing these UDFs, existing complex analytical pipelines can be moved inside the database. This allows us to gain all the advantages of storing data inside a relational database, while still having flexible and easy-to-use analytical tools available.

An additional benefit of training and using machine learning models directly in the database is that it is possible to persist both models and metadata (e.g. classification scores on test sets) in the database. Standard relational queries can then be used to apply the trained models to data. This allows for example to compare and combine output from multiple models, each specialized for certain classification tasks. Also, it is possible to classify the same data using multiple models and use the result of the model that reports the highest confidence.

In this chapter, we showcase how we can use MonetDB/Python UDFs to efficiently integrate a complex analysis pipeline inside MonetDB. We show how we can train models directly inside the database, and how to store the models and subsequently use

them to classify data without having to export the data from the database system.

### 1.1 Contributions

In this chapter, we show how traditional classification models can be integrated into a column-store relational database management system. We describe how models can be stored inside the database system and how these models can then be used to efficiently and flexibly classify data. We experimentally show the performance benefit of directly running the models inside the database system versus loading the data from structured text, binary files or using database client protocols.

### 1.2 Outline

This chapter is organized as follows: Section 2 discusses related work. Section 3 presents our integration approach, followed by a concrete use-case and performance results in Section 4. Finally, we draw our conclusions in Section 5.

## 2 Related Work

There is a variety of related work on combining relational database systems with machine learning pipelines. In this section we will present the most recent related work regarding the integration of machine learning through UDFs and model management systems and compare them with our solution.

### 2.1 Machine Learning Integration

Integrating existing Database Management Systems and machine learning algorithms has been a long standing problem due to the complexity of implementing the machine learning code inside a DBMS.

Early work [73, 3] on this focuses on rewriting analytical algorithms into portable SQL code. This allows the pipelines to be executed within any database system without requiring database-specific modifications. However, rewriting complex analytical

pipelines in SQL requires a lot of manual effort and might not be possible for certain algorithms because SQL is not a Turing complete language.

In Ordonez et al. [63], machine learning algorithms are translated to either C, C++ or C# code (depending on the DBMS language support) and inserted into UDFs. As a consequence they achieve high performance when analyzing large data sets compared to external data analysis tools, as data movement is mitigated. However, these algorithm must be coded in one of the previously listed languages. This often results in the need for rewriting code, because most prominent machine learning libraries are usually available in scripting languages (e.g., Python and R). In our solution we allow the developer to use popular scripting languages *together with their entire ecosystem of data analytics packages* as UDFs in MonetDB.

Other work [26, 17, 37] focuses on more templated approaches for machine learning integration to reduce the necessity of code rewriting. However, the main disadvantage of these methods is that they only work for a limited subset of algorithms, which limits their applicability to general machine learning tasks.

## 2.2 Machine Learning Model Management

When training and using a variety of models the problem of managing these models arises. This problem is exasperated because most Machine Learning Systems do not provide support for storing and querying their models. Due to these issues, data scientists quickly lose track of their models.

In Vartak et al. [85], a system called ModelDB is introduced that can be used for storing, tracking and managing machine learning models in their native environment. This allows data scientists to use SQL to query their models based on their metadata (e.g., hyperparameters, parameters) and quality metrics (e.g., accuracy). It also has the option to store the used train/test data sets for each model. However, since ModelDB only stores the models in their native environment, it does not provide a solution for coupling machine learning applications with traditional relational databases.

# 3 Machine Learning integration

Machine learning pipelines consist of three stages [21].

1. **Preprocessing.** In this stage, the raw data is loaded and cleaned. The data is normalized, and any inconsistencies from incorrect or missing measurements are corrected for or removed.

2. **Training and Verification.** In this stage, the cleaned data is used to train the model. Typically the training set is divided into parts, and techniques like cross validation are used to prevent overfitting the model.

3. **Classification.** In the final stage, the trained model is used to classify new data. In this stage, the model can still be refined further based on new data or new properties of the data.

The preprocessing stage can often be performed entirely within traditional database management systems. Loading data and simple cleaning operations such as missing value removal can be done using standard SQL queries. However, when more advanced preprocessing such as interpolation is required, user-defined functions can be used to simplify this step.

The real challenge of integrating these pipelines into databases, however, is implementing the machine-learning models. The models rely on complex math operations and iterative refinement, which are not supported by standards-complaint SQL.

There are many libraries and packages in vectorized scripting languages that implement common machine learning and classification models, such as TensorFlow [2] and Sci-Kit Learn [65]. Using vectorized user-defined functions, we can plug these libraries into the database. However, the typical processing pipelines must be adjusted so they can fit into a SQL workflow. In this section, we will describe how these analytical pipelines can be integrated into traditional database management systems through the use of user-defined functions.

## 3.1 Training

To train a classification model, we take a set of annotated data as input and use the annotations to find patterns in the data. After learning these patterns, the trained model can accurately classify un-annotated data.

The training pipeline therefore takes as input a set of columns representing the data, and a single column representing the classes of the data. This will be the input to our user-defined function. The output of this stage of the pipeline is the trained model, which will be the output of our UDF. The actual creation and training of the model will happen inside the function.

**Model Storage.** Models exist as in-memory objects within the scripting language. However, they can be serialized to a binary format for persistent storage on disk. In Python, this is done using the `pickle` library. In order to store the objects in the database we need to serialize the objects to this binary format, after which we can place them in a `BLOB` field.

```
CREATE FUNCTION train(data INTEGER, classes INTEGER,
    n_estimators INTEGER)
RETURNS TABLE(classifier BLOB, estimators INTEGER)
LANGUAGE PYTHON
{
    import pickle
    from sklearn.ensemble
        import RandomForestClassifier

    clf = RandomForestClassifier(n_estimators)

    clf.fit(data, classes)

    return {'classifier': pickle.dumps(clf),
        'estimators':n_estimators }
};
```

Listing 5.1: Training The Model

An example of a user-defined function that trains a Random Forest Classifier using Sci-Kit Learn is given in Listing 5.1. This is a vectorized user-defined function, and as such both `data` and `classes` are vectors of integers within the function instead of individual elements. This function can be called from within SQL with the model data, classes and the amount of estimators (i.e., model parameters) as input, and will produce a table containing the trained classifier and its meta-data as output. This table can either be stored in the database, or used directly as input to another function that uses the trained classifier (if no persistent storage is necessary). Note that it is trivial to alter this UDF to train a different model from the Sci-Kit Learn library, as all that is required is importing a different model and using that.

## 3.2 Classification

After the model has been trained, it is ready to accept unlabeled data and can be used to classify that data. The classification stage therefore takes as input a set of columns representing the unannotated data, and the trained classifier that will be used to classify the data. The output is the set of predicted labels produced by the classifier. Inside the user-defined function, the classifier will again have to be deserialized into an in-memory object, after which it can be used to classify the input data and produce a set of labels.

```
CREATE FUNCTION predict(data INTEGER, classifier BLOB)
RETURNS INTEGER
LANGUAGE PYTHON
{
    import pickle
    classifier = pickle.loads(classifier)
    return classifier.predict(data)
};
```

Listing 5.2: Classification

An example of a user-defined function that classifies a set of data is given in Listing 5.2. This function can be called from within SQL with the unlabeled data and

the classifier as input, and will produce a list of predicted classes.

The predict function can be used both to test a trained model and to classify a set of new data using such a model. The model can be tested by predicting a set of data for which the labels are known, and comparing the predicted labels against the new labels. The model can be used to

### 3.3 Ensemble Learning

In addition to only storing the trained models, we can store additional metadata about the models in the database. This metadata can include information such as parameters used to instantiate the model, or information about the effectiveness of the model obtained through testing it against certain datasets. We can then choose a model to classify new data based on this metadata, or we could classify the data using multiple models that are stored and use the results from the classifier with the highest confidence.

## 4 Experimental Analysis

In this section, we demonstrate how a real classification pipeline can be integrated into a column-store database, and show how the in-database processing pipeline performs when compared against the same pipeline implemented in a standard scripting language where the input data is loaded from a file or transferred over a database socket connection.

The pipeline we use in our experiments is used to attempt to classify who people from North Carolina will vote for in the Presidential Elections based on data from the 2012 Presidential Election. For this purpose, we use two separate datasets:

- **The North Carolina Voters Dataset** contains the information about the individual voters. This is a dataset of 7.5M rows, where each row contains information about the voter. There are 96 columns in total, describing characteristics such as place of residence, gender, age and ethnicity. Note that we do not know

who each person actually voted for, as this information is not publicly available.

- **The Precint Votes Dataset** contains the aggregated voting statistics for each precinct, (i.e., how many people in each precinct voted Democrat, and how many voted Republican). This dataset has 2751 rows, one for each precinct in North Carolina.

By combining these two datasets we can attempt to classify individual voters. We know the voting records of a specific precinct, and we know in which precinct each person voted, so we can make an educated guess who each person voted for based on this information.

**Preprocessing.** As we do not have the true class labels for each voter, we have to generate them from the information we have about the precincts. This requires us to join the voter data with the precinct data, giving us the voting records of the precinct that each voter voted in. We then generate a "true" class label for each voter using a weighted random function based on the precinct voting records. For example, if voters in a specific precinct voted for Democrats 60% of the time, each voter in that precinct has a 60% chance of being classified as Democrat and 40% chance of being classified as Republican.

**Training.** After we have generated the true class labels, we have to train the model using the data and the labels. However, we don't simply want to use all the data for training. Instead, we want to divide the data into a training set and a test set to prevent overfitting. We then feed the data in the training set to the model using the function shown in Listing 5.1 and store the resulting model in the database.

**Testing.** After the model is trained, we want to test how it performs by classifying the data in the test set and looking at the results. We can classify the voters in the test set by running the function shown in Listing 5.2. After having obtained the predicted class labels, we can test the accuracy of our model by comparing against the known true class labels of the data. However, since we only have the generated class labels of the individual voters, comparing the predicted labels against those would not give us a lot of information about our classification accuracy. Instead, we aggregate

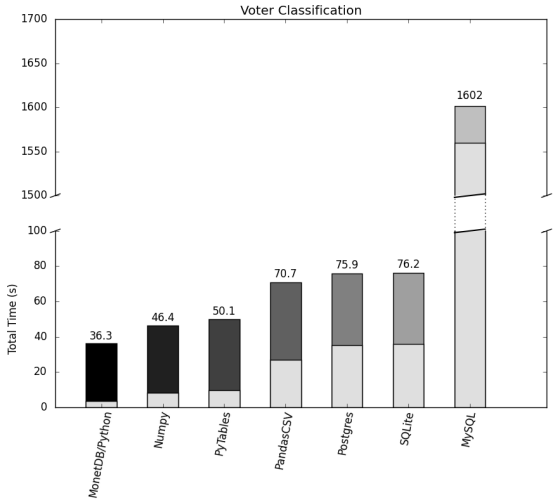



Figure 5-1: Voter Classification Benchmark

the total amount of predicted votes for each party by precinct. Then we compare the aggregated predictions against the known amount of votes in each precinct.

**Performance Analysis.** To determine how well our in-database processing solution performs compared to ad-hoc analysis pipelines we have implemented the pipeline described above both (1) using MonetDB/Python UDFs and (2) inside Python, using various different methods of initially loading the data. For loading the data in Python, we have experimented with loading from binary files (NumPy [84] files and HDF5 [80] using PyTables), CSV files using an optimized parser, transferring the data to Python through a database socket connection (with PostgreSQL [77], MySQL [89] and SQLite [4] as database servers). For the scenarios where the data is stored inside a relational database, we use SQL to perform the preprocessing steps involving joins and aggregations. Whereas for the pure Python solutions, we use the Pandas library [56] to perform these steps.

The experiments were run on a Fedora (Release 26) machine with 2.6GHz 8-core Intel Xeon processor (Turbo Boost up to 3.2GHz), 20MB shared L3 cache and 256 GB of RAM. All the tests are hot runs. The datasets and source code used for the experiments are publically available[1].

[1] `https://github.com/pholanda/VoterClassification`

**Results.** The results of the benchmark are displayed in Figure 5-1. The numbers display the total time required to run the entire classification pipeline, whereas the bottom gray bars indicate the time spent loading the initial data into Python and performing the initial preprocessing steps and aggregations.

We can see that the in-database processing solution using MonetDB/Python is significantly faster than the alternative database solutions. The time spent on initial wrangling of the data is an order of magnitude lower than transferring it over a socket connection using the other database solutions. We also note that loading the data from CSV files is comparable in speed to transferring the data over a socket connection.

Loading the data from binary files is much faster than loading from structured text or transferring the data over a socket connection. However, this introduces additional challenges in managing the data. Especially in the case of NumPy binary files, where each of the 96 columns is stored as a separate file on disk. We do still see that the in-database processing solution spends less time on initial wrangling of the data and runs the entire pipeline significantly faster.

# 5 Summary

In this work, we have shown how complex analysis pipelines can be efficiently integrated into column-store databases. Using these pipelines, it is possible to perform preprocessing, training, testing and prediction using complex machine learning models directly on data stored within a relational database. We have demonstrated the efficiency gained from using these in-database processing methods, and shown the additional benefits that come with storing data in a relational database system.

CHAPTER 6

# MonetDBLite

## 1 Introduction

In Chapter 2, we described an alternative method of combining database management systems and external programs: embedding the database inside the client program. This method has the advantage that the database server no longer needs to be managed, and the database can be installed from within the standard package manager of the tool. In addition, because the database and the analytical tool run inside the same process, data can be transferred between them for a much lower cost.

SQLite [4] is the most popular embedded database. It has bindings for all major languages, and it can be embedded without any licensing issues because its source code is in the public domain. However, it is first and foremost designed for transactional workloads on small datasets. While it can be used in conjunction with popular analytical tools, it does not perform well when used for analytical purposes.

In this chapter, we describe MonetDBLite, an Open-Source embedded database based on the popular columnar database MonetDB [41]. MonetDBLite is an in-process analytical database that can be run directly from within popular analytical tools

without any external dependencies. It can be installed through the default package managers of popular analytical tools, and has bindings for C/C++, R, Python and Java. Because of its in-process nature, data can be transferred between the database and these analytical tools at zero cost. The source code for MonetDBLite is freely available[1] and is in active use by thousands of analysts around the world.

### 1.1 Contributions

We describe the internal design of MonetDBLite, and how it interfaces with standard analytical tools. We discuss the technical challenges we have faced in converting a popular Open-Source database into an in-process embeddable database. We benchmark MonetDBLite against other alternative database systems when used in conjunction with analytical tools, and show that it outperforms alternatives significantly. This benchmark is completely reproducible with publicly available source code.

### 1.2 Outline

This chapter is organized as follows. In Section 2, we describe the design and implementation of the MonetDBLite system. We compare the performance of MonetDBLite against other database systems and statistical libraries in Section 3. Finally, we draw our conclusions in Section 4.

## 2 Design & Implementation

In this section we will discuss the general design and implementation of MonetDBLite, and the design choices we have made while implementing it.

### 2.1 Internal Design

MonetDBLite is based on the popular Open-Source columnar database MonetDB, and as such it shares most of its internal design. The core design of MonetDB is described

[1] `https://github.com/hannesmuehleisen/MonetDBLite`

in Idreos et al. [41]. However, since this publication a number of core features have been added to MonetDB. In this section, we give a brief summary of the internal design of MonetDB and describe the features that have been added to MonetDB since.

## 2.2 Embedding Interface

MonetDBLite is a database that is embedded into analytical tools directly, rather than running as a standard client-server database. As MonetDBLite runs within a process, clients have to create and initialize the database themselves rather than connecting to an existing database server through a socket connection. For this purpose, MonetDBLite needs a set of language bindings so the database can be initialized and queries can be issued to the database.

MonetDBLite has language bindings for the C/C++, R, Python and Java programming languages. However, all of these are wrappers for the C/C++ language bindings. The main challenge in creating these wrappers is converting the data to and from the native types of each of these languages. The optimization challenges of this type conversion are discussed in Section 2.3. In this section, we will discuss only the C/C++ API.

The database can be initialized using the `monetdb_startup` function. This function takes as optional parameter either a reference to a directory in which it can persistently store any data. If no directory is provided, MonetDBLite will be launched in an in-memory only mode, in which case no persistent data is saved to disk.

If the database is launched in persistent mode, a new database will be created in the specified directory if none exists yet. Otherwise, the existing database will be loaded and potentially upgraded if it was created by an older version of MonetDBLite. If the database is launched in-memory, a new temporary database will be created that will be kept entirely in-memory. Any data added to the database will be kept in-memory as well. After an in-memory database is shut down, all stored data will be discarded. The regular MonetDB does not have this feature.

After a database has been started, connections to the database can be created using the `monetdb_connect` function. In the regular MonetDB server, these connections

represent socket connections to a client process. In MonetDBLite, however, these connections are dummy clients that only hold a query context and can be used to query the database. Multiple connections can be created for a single database instance. These connections can be used for inter-query parallelism by issuing multiple queries to the database in parallel and they provide transaction isolation between them.

Using these connections, the embedded process can issue standard SQL queries to the database using the `monetdb_query` function. This function takes as input a client context and a query to be issued, and returns the results of the query to the client in a columnar format in a `monetdb_result` object. The `monetdb_result` object is semi-opague, exposing only a limited amount of header information, as shown in Listing 6.1

```
1 struct monetdb_result {
2   size_t nrows;
3   size_t ncols;
4   char type;
5   size_t id;
6 };
```

Listing 6.1: MonetDBLite Result Object

The individual columns of the result can be fetched using the `monetdb_result_fetch` function, which takes as input a pointer to the `monetdb_result` object and a column number. There are two versions of this function: a low level version, and a high level version. In the low level version, the underlying structures used by the database are directly returned without any conversions being performed. This function requires internal knowledge of the database internals, and is intended for use primarily for the language-specific wrappers for extra performance. In the high level version, the database structures are converted into a set of simple structures that can be used without knowledge of the internals of MonetDB(Lite). The returned structures depend on the type of the column. An example for the `int` type is given in Listing 6.2.

```
struct monetdb_column {
    monetdb_type type;
    int* data;
    size_t count;
    int null_value;
    double scale;
    int (*is_null)(int value);
};
```

Listing 6.2: MonetDBLite Integer Column

In addition to issuing SQL queries, the embedded process can efficiently bulk append large amounts of data to the database using the `monetdb_append` function. This function takes the schema and the name of a table to append to, and a reference to the data to append to the columns of the table. This function allows for efficient bulk insertions, as there is significant overhead involved in parsing individual `INSERT INTO` statements, which becomes a bottleneck when the user wants to insert a large amount of data.

## 2.3 Native Language Interface

For any of the languages other than C/C++, data has to be converted between the database's native format to the target language's native format. When a SQL query is issued, the result has to be mapped back into the target environment. Likewise, if the user wants to move data from the target environment to the database, it has to be converted.

Database connectors in the target environment face a similar but more difficult problem, as they also have to deal with communicating with the remote database server. We could adapt these database connectors to work with MonetDBLite. However, in an analytical context this approach is problematic. As these are row-focused interfaces [70], the results of queries must be fetched one-by-one. This leads to a large amount of overhead when fetching a large result set, especially in interpreted scripting languages such as R or Python. Columnar bulk access to result sets is therefore

needed, where all values belonging to one column can be fetched into a set of arrays, one per column, in one or few calls to the database interface.

However, not all arrays are created equal. While it is possible to subclass the native array representation in most programming environments, efficiency concerns and expectations by third-party software might make a fully native data representation necessary. For example, in R, most third-party packages will contain some portion of compiled code written in C/C++, which relies on arrays being stored in the native bit representation if they are to compute anything meaningful with them. Similarly, in the NumPy environment, third-party packages can get a pointer to the native C representation of any array. Hence for the objects that we return from the database to be able to be used by these packages, we must create objects that exactly match the native array format of the target environment.

**Zero-Copy.** Every target environment has a particular array representation in memory. However, due to hardware support contiguous C-style arrays are ubiquitous for numerical values. For example, both R and NumPy use this representation to store arrays of numerical data. This allows for a unique optimization opportunity: Instead of converting the data into a freshly allocated memory area, we can choose to share a pointer to the existing data with the target system. The memory layout needs to be compatible between data management and target environment, e.g. both using contiguous C-style arrays containing four-byte signed integers. If this pointer sharing is possible, the only cost comes from initializing metadata structures in the target environment (e.g. SEXP header in R). However, this cost does not depend on the size of the data set.

Great care needs to be taken to prevent modification of the data being shared. The target environment may run any program imaginable, including code from contributed packages, that may try to modify the shared memory areas. The shared pointer might be part of persistent data of the database, hence modifying the data directly could lead to corruption of the data stored in the database. Because of this, no direct modification of this data is allowed.

What is desirable here are copy-on-write semantics for the shared data. If code

from the target environment attempts to write into the shared data area, the data should be copied within the target environment and only the copy modified. To ensure these semantics are enforced, the Unix `mprotect` kernel function can be used to disallow writes to the data by the target code. When the target environment attempts to modify data we have shared with it, we create copy and modify the copy instead. This allows for efficient read-only access without the risk of data corruption.

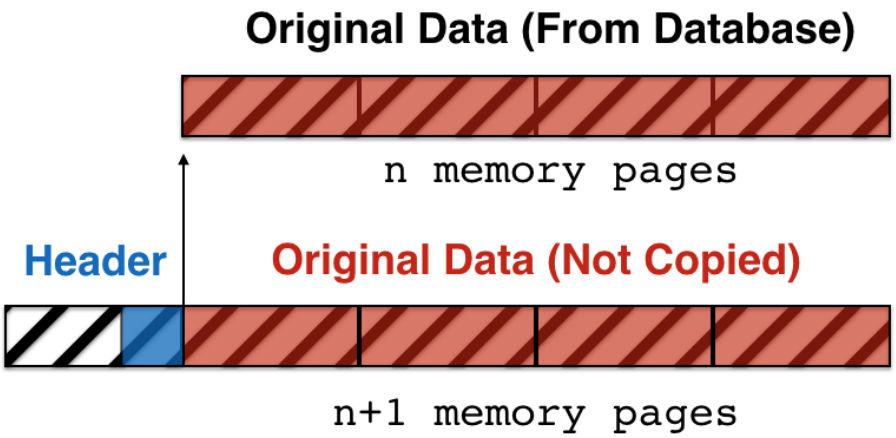



Figure 6-1: Header forgery for zero-copy data transfer.

**Header Forgery.** A challenge of providing a zero-copy interface to the data stored in the database is that certain libraries expect metadata to be stored as a header physically in front of the data. This is accomplished in the library by performing a single memory allocation that allocates the size of the header plus the size of the data. This is problematic in our scenario. As the source data comes directly from an external database system it does not have space allocated in front of it for these headers.

This problem could be solved by making the database always allocate extra bytes in front of any data that could be passed to the analytical tool. As we have full control of the database system, this is feasible. However, it would require a significant amount of code modification and would result in wasted space in scenarios where the data is not passed to the analytical tool.

Instead, we solve this problem using header forgery. This process is shown in Figure 6-1. To provide a zero-copy interface of a region of `n` memory pages, we allocate

a region of size $n + 1$ memory pages using the `mmap` [28] function. We then place the header information at the end of the first page. We then use the `mmap` function together with the `MAP_FIXED` flag to directly link the remaining `n` memory pages to the original data. This linking happens in the memory page table, and does not create a copy of these pages. This method predates, but could be considered an application of the memory rewiring technique presented in [74].

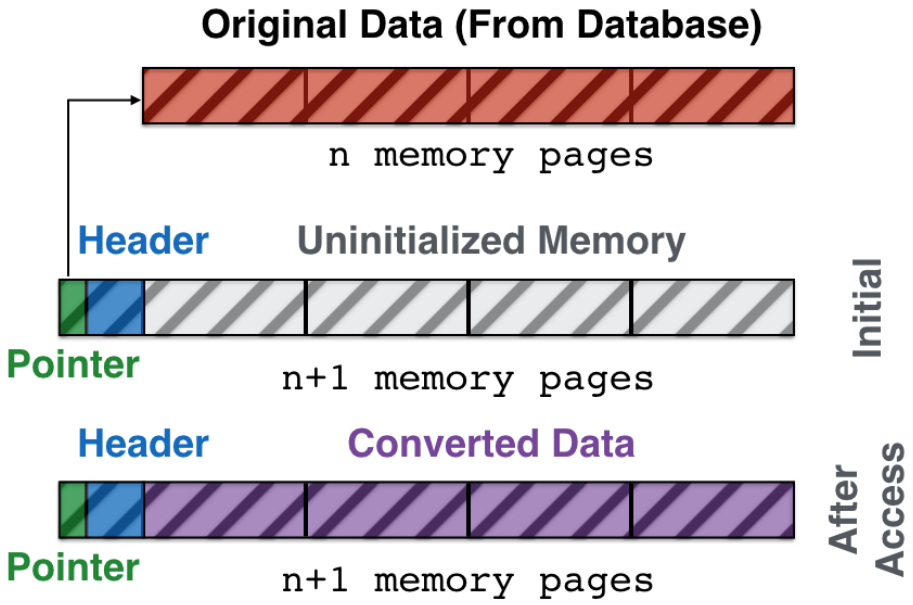



Figure 6-2: Lazy data conversion.

**Lazy Conversion.** While the zero-copy approach is ideal, as it does not require us to touch the to–be–converted data, it cannot be used in all cases. When the internal representation of the database is not bit-compatible with that of the target environment, data conversion has to be performed. As all data has to be converted, the conversion will take a linear amount of time w.r.t. the size of the result set. However, it is not known whether the target environment will ever actually do anything with the converted data. It is not uncommon for a user to perform a query such as `SELECT * FROM table` and only access a small amount of columns from the result.

This issue can be resolved by performing lazy conversion of the result set. Instead of eagerly converting the entire result set, we create a set of "dummy" arrays that start out with a correctly initialized header. However, the data is filled with uninitialized

memory. This is shown in Figure 6-2. We then use the `mprotect` [29] function to protect the uninitialized memory from being read or written to directly using the `PROT_NONE` flag. When the user attempts to access the protected memory area, the system throws a segmentation fault, which we then catch using a signal handler. Using a pointer to the original data that is stored alongside the header, we then perform a conversion of the actual data and unset the `mprotect` flag, allowing the user to use the now-converted data transparently.

## 2.4 Technical Challenges

In this section, we will discuss the additional technical challenges that we encountered while converting a standard relational database management system to an in-process embedded database system.

**Internal Global State.** MonetDB was originally designed to run as a single stand-alone process. One of the consequences of this design is that internal global state (global variables) is used often in the source code. The database uses global state to keep track of e.g. the data stored inside the database, the write-ahead logger and numerous database settings.

This global state leads to a limitation: it is not possible to run MonetDBLite twice in the same process. As the global state holds all the information necessary for the database to function, including paths to database files, and this information is continuously accessed while the database is running, only one database server can be running in the same process. To make it possible to run several database servers within the same process would require a very comprehensive code rewrite, as the global database state would have to be passed around to almost every function.

**Garbage Collection.** Another issue caused by this global state is garbage collection. As the database server no longer runs as a stand-alone program, the global variables can no longer be reset by restarting the server. In addition, all the allocated memory has to be freed in the process. Allocated regions can no longer be neglected with the knowledge that they will be freed when the process is terminated. Instead, to properly support an "in-process shutdown" of the database server, everything has to

be cleaned up manually and all global variables have to be reset to their initial state.

**External Global State.** Another consequence of the database server being designed to run as a stand-alone process is that it modifies a lot of external global state, such as signal handlers, locale settings and input/output streams. For each of these, it was necessary to modify the database source code to not modify the global state. Otherwise loading the database package would result in it overriding signal handlers, leading to e.g. breaking the scripting languages' input console.

Calls to the `exit` function were especially problematic. In the stand-alone version of MonetDB the database server shuts down when a fatal error was detected (such as running with insufficient permissions or attempting to open a corrupt database). This happens mostly during start-up. This is expected behavior in a stand-alone database server, but becomes problematic when running embedded inside a different program. Attempting to access a corrupt database using the embedded database would result in the entire program crashing, rather than a simple error being thrown. Even worse, since the database would simply exit in these scenarios, no alternative path exists to only report the error. To avoid a large code rewrite, we used `longjmp` whenever the `exit` function was called, which would jump out of the `exit` and move to a piece of code where the error could be reported.

**Error Handling.** Another aspect of the database design that we needed to rethink was error handling. In the regular database server, errors are reported by writing them to the output stream so they can be handled by the client program. However, in the embedded version the errors must be reported as a return value from the SQL query function. We had to rewrite large portions of the error reporting code to accommodate this.

**Dependencies.** To make MonetDBLite as simple to install as possible, one of our design goals was to remove all external dependencies. Regular MonetDB has a large number of required dependencies, among which are `pcre`, `openssl`, `libxml` and `pkg-config` along with a large number of optional dependencies. For MonetDBLite, we stripped all of these dependencies by removing large chunks of optional code and rewriting code that relied on any of the required dependencies. For example, we made

our own implementation of the `LIKE` operator (that previously used regular expressions from the PCRE library). As a result of our efforts, MonetDBLite has no external dependencies and can be installed without having to install any other libraries.

# 3 Evaluation

In this section, we perform an evaluation of the performance of MonetDBLite and compare it against both (1) other relational database management systems, and (2) several popular RDBMS alternatives used in statistical tools.

## 3.1 Setup

All experiments in this section were run on a desktop-class computer with an Intel i7-2600K CPU clocked at 3.40GHz and 16 GB of main memory running Fedora 26 Linux with Kernel version 4.14. We used GCC version 7.3.1 to compile systems. Reported timings are the median of ten hot runs. The initial cold run is always ignored. A timeout of 5 minutes is used for the queries.

**Systems.** The following systems were used to compare against in our benchmarks. All systems were configured to only use one of the eight available hardware threads for fairness (as not all systems support intraquery parallelism). Furthermore, unless indicated otherwise, we have attempted to configure the systems to take full advantage of available memory. The complete configuration settings and scripts to reproduce the results reported below can be found in the benchmark repository[2].

- **SQLite** [4] (Version 3.20.1) is an embedded SQL database designed for transactional workloads.
- **MonetDB** [41] (Version 11.29.3) is an analytical column-store database.
- **PostgreSQL** [77] (Version 9.6.1) is a row-store database designed for transactional workloads.

[2] `https://github.com/Mytherin/MonetDBLiteBenchmarks`

- **MariaDB** [89] (Version 10.2.14) is a row-store database designed for transactional workloads. It is based on the popular MySQL database.

**Libraries.** In addition to the above-mentioned database management systems, we test the following analytical libraries that emulate database functionality. We only use these libraries in the query execution benchmarks.

- **data.table** [22] (Version 1.11.0) is an R library for performing common database operations.
- **dplyr** [88] (Version 0.7.4) is an R library for performing common database operations.
- **Pandas** [56] (Version 0.22.0) is a Python library for performing common database operations.
- **Julia** [7] (Version 0.6.2) is a JIT compiled analytical language that has support for performing standard database operators through the `DataFrames.jl` library.

**Datasets.** We perform benchmarks using the following data sets.

- **TPC-H Benchmark.** [82]. This synthetic dataset is designed to be similar to real-world data warehouse fact tables. In our benchmarks, we use the scale factors 1 and 10. The scale factor indicates approximately the size of the dataset in GB.
- **American Community Survey (ACS)** [10]. This dataset contains millions of census survey responses. It consists of 274 columns.

## 3.2 TPC-H Benchmark

As we focus on the integration of analytical tools with an analytical database, there are three different scenarios that we want to optimize for and that we will benchmark.

1. **Data Ingestion.** The rate at which data can be imported into the database from the analytical tool. We call this the data ingestion or data import rate. This scenario occurs when users want to take data that is the result of computations in the analytical tool and store it persistently in the database.

2. **Data Export.** The rate at which data can be imported into the analytical tool from the database. This data export rate is important when the user wants to perform analytics on data that is stored persistently within the RDBMS.

3. **Query Execution.** The performance of the database engine when performing analytical queries. This scenario occurs when the user wants to perform operations and aggregations on large amounts of data using the databases' storage engine. Note that for query execution, it is also possible to simply move the data from the database into the analytical tool and do the processing there using the previously mentioned libraries. For that reason, we also compare the performance of the RDBMS with the afore-mentioned libraries.

## Data Ingestion

For the data ingestion benchmark, we only consider the `lineitem` table. This is the biggest table in TPC-H. It has 16 columns, primarily of types `DECIMAL`, `DATE` and `VARCHAR`. There are no `NULL` values.

For this experiment, we read the entire `lineitem` table into R and then use the `dbWriteTable` function of the R DBI [87] API to write the table into the database. After this function has been completed, the table will be persistently present within the database storage engine and all the data will have been loaded into the database. We only consider the database systems for this experiment.

The results of this experiment can be seen in Figure 6-3. We can see that MonetDBLite has the fastest data ingestion. However, we note that SQLite is not very far behind MonetDBLite. For both systems, the primarily bottleneck is writing the data to disk. MonetDBLite gains performance by storing the data in a more compact columnar format, rather than the B-tree structure that SQLite uses to store data

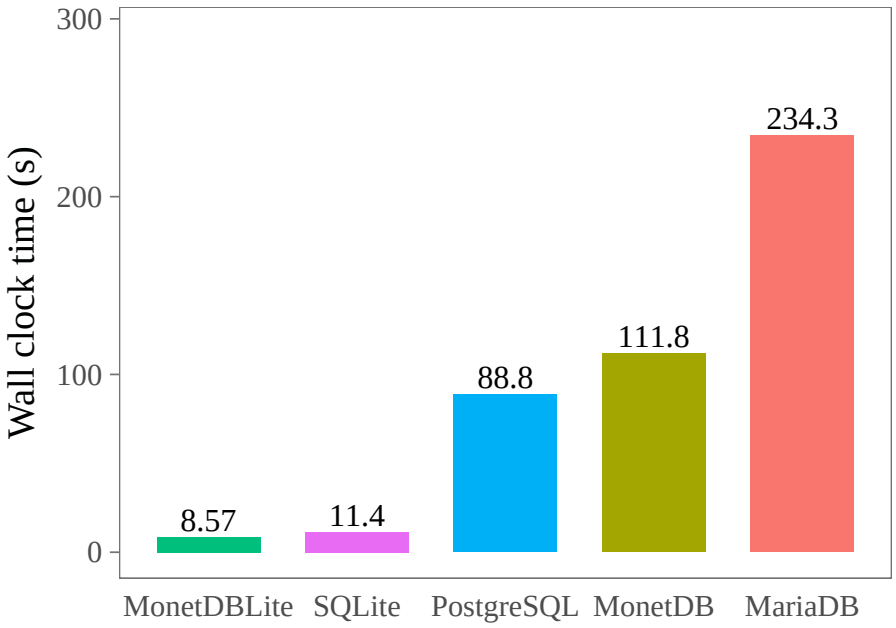



Figure 6-3: Writing the `lineitem` table from R to the database.

internally.

All the other systems perform extremely poorly on this benchmark. This is because the data is written to the database over a socket connection, which requires a large amount of network communication. However, the main problem is that these database systems do not have specialized protocol code for copying large amounts of data from the client to the server console. Instead, the data is inserted into the database using a series of `INSERT INTO` statements, which introduces a large amount of overhead leading to orders of magnitude worse performance than the embedded database systems.

## Data Export

For the data export benchmark, we again only consider the `lineitem` table of the TPC-H benchmark. For this experiment, we read the entire `lineitem` table from the database into R using the `dbReadTable` function of the R DBI. This effectively performs a `SELECT * FROM lineitem` query on the database and stores the result of this query inside an R data frame.

The results of this experiment can be seen in Figure 6-4. We can see that

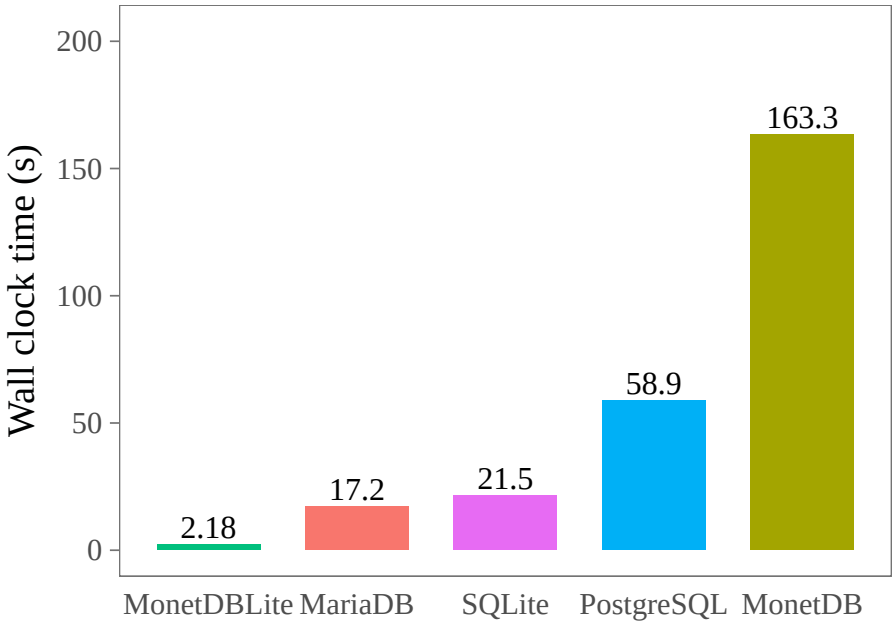



Figure 6-4: Loading the `lineitem` into R from the database.

MonetDBLite has by far the fastest data export rate. Because it runs within the analytical process itself, and because it makes use of zero-copy data transfer of numeric columns, the data can be transferred between the database system and R for almost no cost. By contrast, the databases that are connected through a socket connection take a significantly longer time to transfer the result set to the client.

Despite running in-process as well, SQLite also takes a very long time to transfer data from the database to the analytical tool. This is because the conversion of data from a row-major to column-major format takes a significant amount of time.

## Query Execution

For the query execution benchmark, we run the first ten queries of the TPC-H benchmark inside each of the systems. For each of the libraries, we have created an equivalent script for each of the queries using each of the libraries.

**Library Implementations.** Note that, since the libraries naively execute user code without performing any high-level strategic optimizations, there is a lot of room for modifying their performance as the equivalent functionality could be implemented

TPC-H SF 1

| System | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 |
|---|---|---|---|---|---|---|---|---|---|---|
| MonetDBLite | 0.74 | 0.03 | 0.29 | **0.06** | **0.07** | 0.18 | **0.08** | **0.08** | **0.09** | 0.20 |
| MonetDB | 0.87 | **0.02** | **0.09** | 0.08 | 0.10 | **0.05** | 0.08 | 0.11 | 0.16 | **0.07** |
| SQLite | 8.41 | 0.04 | 1.83 | 0.44 | 1.00 | 1.17 | 6.52 | **T** | 19.05 | 1.35 |
| PostgreSQL | 8.93 | 0.25 | 0.71 | 2.08 | 0.46 | 1.06 | 0.62 | 0.60 | 2.31 | 1.40 |
| MariaDB | 19.65 | 1.96 | 4.87 | 0.97 | 4.16 | 2.02 | 2.13 | 6.71 | 18.12 | 15.67 |
| data.table | **0.45** | 0.12 | 0.28 | 0.20 | 0.46 | 0.13 | 0.27 | 0.24 | 0.88 | 0.20 |
| dplyr | 0.70 | 0.13 | 0.34 | 0.25 | 0.60 | 0.17 | 0.31 | 0.41 | 1.17 | 0.28 |
| Pandas | 0.85 | 0.19 | 0.49 | 0.41 | 0.93 | 0.12 | 0.44 | 0.56 | 1.82 | 0.34 |
| Julia | 0.99 | 0.10 | 0.73 | 0.25 | 0.53 | 0.07 | 0.30 | 0.67 | 1.05 | 0.57 |

TPC-H SF 10

| System | Q1 | Q2 | Q3 | Q4 | Q5 | Q6 | Q7 | Q8 | Q9 | Q10 |
|---|---|---|---|---|---|---|---|---|---|---|
| MonetDBLite | 16.55 | 0.14 | 1.92 | **0.50** | **0.64** | 0.44 | **0.68** | **0.75** | **0.95** | 0.99 |
| MonetDB | **9.63** | **0.07** | **1.15** | 0.87 | 1.16 | **0.38** | 1.00 | 1.12 | 1.66 | **0.68** |
| SQLite | 97.61 | 0.37 | 23.17 | 4.44 | 12.65 | 11.69 | **T** | **T** | **T** | 14.72 |
| PostgreSQL | 88.77 | 2.71 | 63.87 | 22.87 | 4.92 | 11.41 | 7.68 | 6.73 | 74.42 | 63.54 |
| MariaDB | 169.58 | 20.76 | 124.59 | 13.34 | 78.88 | 33.42 | 88.72 | 139.68 | 218.65 | 234.95 |
| data.table | **E** | **E** | **E** | **E** | **E** | **E** | **E** | **E** | **E** | **E** |
| dplyr | 31.48 | 1.20 | 5.13 | 3.79 | 8.13 | 1.83 | 4.35 | 4.47 | 16.29 | 3.77 |
| Pandas | **E** | **E** | **E** | **E** | **E** | **E** | **E** | **E** | **E** | **E** |
| Julia | 24.61 | 5.00 | 7.32 | 2.78 | 9.51 | 0.66 | 7.32 | 13.42 | 18.90 | 6.14 |

Table 6.1: Performance Results for TPCH SF1 and SF10

in many naive and inefficient ways. In the worst case, we could perform cross products and filters instead of performing standard joins. Likewise, we could choose poor join orders or not perform filter or projection push down, and force materialization of many unused tuples.

To attempt to maximize the performance of these libraries, we manually perform the high-level optimizations performed by a RDBMS such as projection pushdown, filter pushdown, constant folding and join order optimization. We have created these implementations by using the query plans that are executed by VectorWise [9], a state–of–the–art analytical database system that is the front runner on the official TPC-H benchmark for single node machines. All the scripts that we have created for each of the libraries can be found in our software repository. We have also reached out to the developers of each library and received feedback on optimization.

However, having the user apply all these optimizations is not realistic. This scenario

assumes the user has perfect knowledge on how to order joins and assumes the user does not do any inefficient steps such as including unused columns. The benchmark results provided for these libraries should therefore be seen as a *best-case performance scenario*. The benchmark results for these libraries would be significantly worse if we did not manually perform many of the automatic optimizations performed by a database system.

## TPC-H SF1

The total time required to complete all the measured TPC-H queries for the different systems is shown in Table 6.1. We can see that both MonetDB and MonetDBLite show the best performance on the benchmark. They also show very similar performance. This is because the TPC-H benchmark revolves around computing aggregates, and does not involve transferring a large amount of data over the socket connection. As such, the bottleneck is almost entirely the computation performed in the database server. As MonetDB and MonetDBLite use the same internal query execution engine, they have identical performance.

After MonetDB, we can see the various libraries we have tested performing similarly with only a factor two difference between the best and the worst performing library. The fastest library, `data.table`, is heavily optimized for performing efficient relational operations. However, even with the optimizations we have performed on the user code it still cannot reach the performance of an actual analytical database system. This is because the procedural nature of these libraries heavily limits the actual optimizations that can be performed compared to the optimizations that a database can perform on queries issued in the declarative language of SQL. For example, they do not perform late materialization.

The traditional database systems perform significantly worse than the libraries, however. As the TPC-H benchmark is designed to operate on large chunks of a subset of the columns of a table, the row-store layout and tuple–at–a–time processing methods of the traditional database systems perform extremely poorly on this benchmark. We can see that the traditional database systems perform many orders of magnitude worse

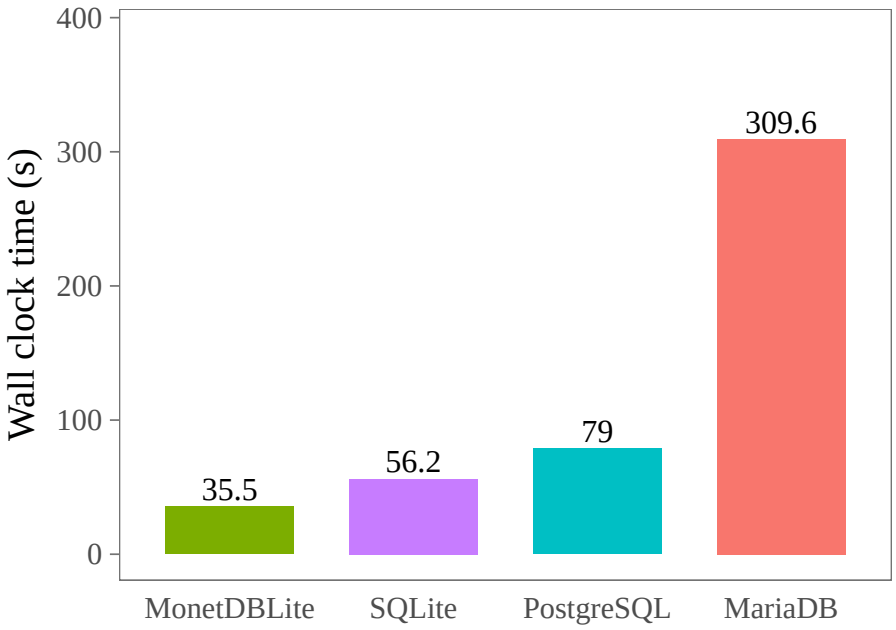



Figure 6-5: Loading the ACS data into the database.

than the analytical database systems and the libraries we have used.

**Individual Query Performance.** The performance of each of the systems on individual queries can be seen in the table as well. The libraries perform extremely well on TPC-H Query 1 and Query 6. On Query 1, `data.table` even manages to beat our analytical database system. The libraries perform well on these queries because the queries only involve performing filters and aggregations on a single table without any joins.

The libraries perform worse on queries involving multiple joins. The join operations in these libraries do not take advantage of meta-data and indices to speed up the joins between the different tables. As such, they perform significantly worse than the analytical database even when using an optimal join order.

The traditional database systems perform poorly on queries that involve a lot of tuples behind pushed through the pipeline to the final aggregations. Because of their tuple–at–a–time volcano processing model they invoke a lot of overhead for each tuple that passes through the pipeline. This results in poor performance when many tuples have to be processed at a time.

### TPC-H SF10

The results for the TPC-H SF10 benchmark are shown in Table 6.1. We note that at this scale factor, the entire dataset still fits in memory. However, each of the scripting libraries run into either out–of–memory errors or heavily penalized performance from swapping on these queries. This is because these libraries require not only the entire dataset to fit in memory, but also require any intermediates created while processing to fit in memory. When the intermediates exceed the available memory of the machine the program crashes with an out–of–memory exception. The database solutions do not suffer from this problem, as they offload unused data to disk using either the buffer pool or by letting the operating system handle it using memory mapped files.

While the traditional database systems do not run into crashes due to running out–of–memory, their performance does degrade by more than an order of magnitude. Because of the row-store layout of these systems, they have to scan and use the entire dataset rather than only the hot columns. As a result, they run into performance penalties as the entire dataset plus the constructed indices do not fit in memory anymore and have to be swapped to disk. The column-store databases do not suffer from this problem because only the actually used columns have to be touched to answer the queries, and these are small enough to be kept in memory.

## 3.3 ACS Benchmark

For the American Community Survey benchmark, we run the ACS survey analysis script as provided by Anthony Damico [20]. The script in this benchmark wrangles data of the American Community Census, a large scale census performed in the United States that gathers data about roughly 1% of the US population every year.

The script consists of two phases. In the first phase, the required data is gathered and downloaded from the official data repositories. In the second phase, the downloaded data is then processed and stored persistently in a database server. The persistently stored data can then be analyzed and various aggregations and statistics can be gathered from the data using the survey package [54].

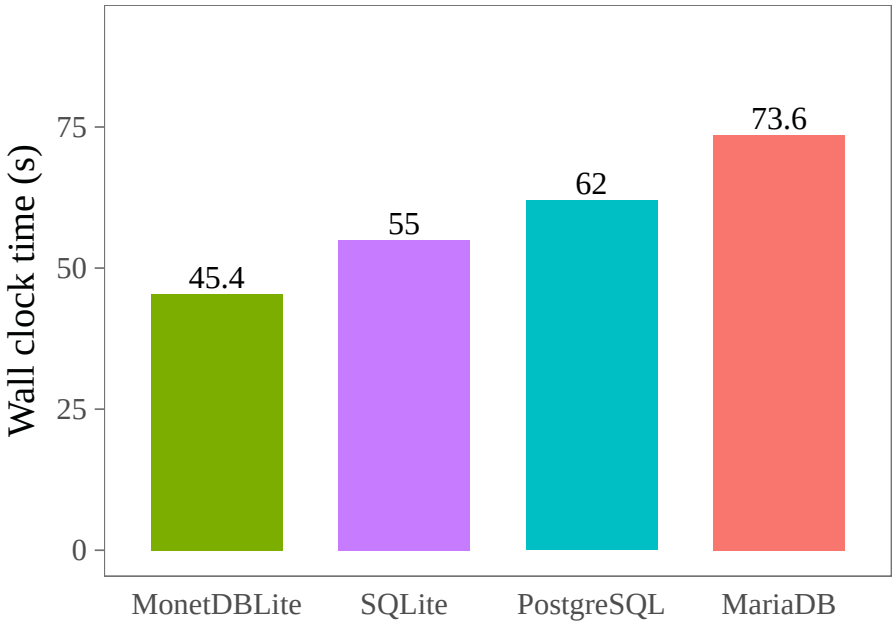



Figure 6-6: Performing the ACS statistical analysis.

The survey package allows you to hook your own database driver into the script, and will perform a significant amount of processing inside the database. For operations were SQL is insufficient, the data is transferred from the database to R and the data is then processed inside R using various statistical libraries.

The official documentation of the ACS script describes a large amount of statistics that can be gathered from the data. For this benchmark, we benchmark both the required loading time into the database (but exclude the time spent on downloading the data) and a number of statistical operations that are described in the official documentation. We limit ourselves to a subset of the data: we only look at the data from five states of the year 2016. This is $\approx 2.5$ GB in data.

## Data Loading

The benchmark results for loading the data in the database are shown in Figure 6-5. MonetDBLite performs the best on this benchmark, but not by as large a factor as seen in the TPC-H benchmark. This is because the survey package performs a lot of preprocessing in R that happen regardless of which database is used. As a result, the

performance difference between the different databases is not as overwhelming but still very visible.

### Statistics

The benchmark results for running the various statistical functions using the different database connectors are shown in Figure 6-6. We can see that the difference between performance of the different database engines is not very large. This is because most of the actual processing happens inside R rather than inside the database. The observed difference in performance is mainly because of the difference in the cost of exporting data from the database. However, since the amount of exported data is not very large compared to the amount of processing that occurs in this scenario there is less than a factor two difference between the systems.

## 4 Summary

In this chapter, we have presented the embedded analytical database system MonetDBLite. MonetDBLite performs orders of magnitude better than traditional relational database systems when executing analytical workloads, and provides an order of magnitude faster interface between the database and the analytical tool.

In addition to being significantly faster, MonetDBLite is also easier to setup and use because it does not require an external server and does not have any dependencies. It can be installed through standard package and library managers of popular analytical tools. All of these factors combined make MonetDBLite highly suitable as a persistent data store for analytical tasks.

# CHAPTER 7

## DuckDB: an Embeddable Analytical Database

## 1 Introduction

In Chapter 6, we described our effort in developing MonetDBLite, an embedded analytical system that is derived from the MonetDB system. MonetDBLite proved successfully that there is a real interest in embedded analytics, it enjoys thousands of downloads per month and is used all around the world from the Dutch central bank to the New Zealand police. However, its success also uncovered several issues that proved very complex to address in a non-purpose-built system. We identified the following requirements for embedded analytical database systems:

- High efficiency for OLAP workloads, but without completely sacrificing OLTP performance. For example, concurrent data modification is a common use case in dashboard-scenarios where multiple threads update the data using OLTP queries and other threads run the OLAP queries that drive visualizations simultaneously.
- Efficient transfer of tables to and from the database is essential. Since both database and application run in the same process and thus address space, there

is a unique opportunity for efficient data sharing which needs to be exploited.

- Controlled resource consumption and ability to operate efficiently on lower-end hardware is essential. While traditional database systems expect to be the sole occupant on a big machine, embedded database systems need to "play nice" with the host application with regards to resource usage.

- High degree of stability, if the embedded database crashes, for example due to an out-of-memory situation, it takes the host down with it. This can never happen. Queries need to be able to be aborted cleanly if they run out of resources.

- Practical "embeddability" and portability, the database needs to run in whatever environment the host does. Dependencies on external libraries (e.g. `openssh`) for either compile- or runtime have been found to be problematic. Signal handling, calls to `exit()` and modification of singular process state (locale, working directory etc.) are forbidden.

MonetDBLite was successful in achieving high efficiency for OLAP workloads and efficient transfer of tables to and from the system. However, the fact that it was designed to be a stand-alone system resulted in many complications that prevented it from being able to fully succeed as an embedded OLAP RDBMS.

The operator–at–a–time processing model used by MonetDB materializes large intermediates entirely in memory. This memory-intensive processing model combined with the fact that MonetDB does not provide hard limits on the amount of memory that it uses can lead to MonetDB quickly consuming all the available memory of the system, leaving no memory left for the host application. This processing model also suffers from performance problems when these intermediates do not fit in memory, as the intermediates will be constantly swapped to disk.

Another issue caused by this processing model is the incapability of interrupting queries in between operators. As every single operator must run completely until the operator is finished, a user cannot quickly interrupt the execution of an expensive operator (e.g. a large cross product). This is especially problematic when the database

is used in interactive scenarios as the user cannot abort a query after realizing that it takes too long to complete.

Additional problems arose from MonetDB's usage of memory mapped files to load the database data from disk. While `mmap` seems like an attractive option to allow the operating system to handle loading of data from disk into memory, the uncontrolled nature of when the data is actually fetched can cause large problems. Whenever any part of a memory mapped region is read, the OS can potentially trigger a load from disk and will send a `SIGBUS` signal to the application if that load fails. As MonetDB passes around memory mapped data all around the processing pipeline, almost any piece of code can trigger a `SIGBUS` signal. While handling these signals is possible in a stand-alone application, it is not possible in a library as signal handlers are process global and can thus (accidentally) be overwritten by the user.

MonetDB also suffers from many practical embeddability problems, including ample usage of global variables, lack of namespacing for function names resulting in potential symbol conflicts, calls to `exit` in case of fatal errors, and reliance on `setlocale` and working directory modification. While these problems can be solved, they require very large rewrites that touch almost the entire codebase.

To tackle these issues, we built *DuckDB*, a new purpose-built embeddable RDBMS. In this chapter, we present the capabilities of DuckDB. DuckDB is available as Open-Source software under the permissive MIT license[1]. DuckDB is no research prototype but built to be widely used, with millions of test queries run on each commit to ensure correct operation and completeness of the SQL interface.

## 1.1 Contributions

We describe the internal design of DuckDB and how it interfaces with standard analytical tools and describe how it tackles the unique challenges that are faced by an embedded analytical database system.

[1] `https://github.com/cwida/duckdb`

| | | |
|---|---|---|
| API | C/C++/SQLite | |
| SQL Parser | `libpg_query` | [27] |
| Optimizer | Cost-Based | [57, 59] |
| Execution Engine | Vectorized | [9] |
| Concurrency Control | Serializable MVCC | [60] |
| Storage | Custom Single-File | |

Table 7.1: DuckDB: Component Overview

# 2 Design and Implementation

DuckDB's design decisions are informed by its intended use case: embedded analytics. Overall, we follow the "textbook" separation of components: Parser, logical planner, optimizer, physical planner, execution engine. Orthogonal components are the transaction and storage managers. While DuckDB is first in a new class of data management systems, none of DuckDB's components is revolutionary in its own regard. Instead, we combined methods and algorithms from the state of the art that were best suited for our use cases.

Being an embedded database, DuckDB does not have a client protocol interface or a server process, but instead is accessed using a C/C++ API. In addition, DuckDB provides a SQLite compatibility layer, allowing applications that previously used SQLite to use DuckDB through re-linking or library overloading.

```
#include "duckdb.hpp"
DuckDB db("/tmp/db.duck");
Connection con(db);
auto result = con.Query("SELECT * FROM tbl");
cout << result->GetValue(0, 0);
```

Listing 7.1: Using DuckDB from C++

The *SQL parser* is derived from Postgres' SQL parser that has been stripped down as much as possible [27]. This has the advantage of providing DuckDB with a full-featured and stable parser to handle one of the most volatile form of its input, SQL queries. The parser takes a SQL query string as input and returns a parse tree of C structures. This parse tree is then immediately transformed into our own parse

tree of C++ classes to limit the reach of Postgres' data structures. This parse tree consists of statements (e.g. `SELECT`, `INSERT` etc.) and expressions (e.g. `SUM(a)+1`).

The *logical planner* consists of two parts, the binder and the plan generator. The binder resolves all expressions referring to schema objects such as tables or views with their column names and types. The logical plan generator then transforms the parse tree into a tree of basic logical query operators such as scan, filter, project, etc. After the planning phase, we have a fully type-resolved logical query plan. DuckDB keeps statistics on the stored data, and these are propagated through the different expression trees as part of the planning process. These statistics are used in the optimizer itself, and are also used for integer overflow prevention by upgrading types when required.

DuckDB's *optimizer* performs join order optimization using dynamic programming [57] with a greedy fallback for complex join graphs [61]. It performs flattening of arbitrary subqueries as described in Nuemann et al. [59]. In addition, there are a set of rewrite rules that simplify the expression tree, by performing e.g. common subexpression elimination and constant folding. Cardinality estimation is done using a combination of samples and HyperLogLog. The result of this process is the optimized logical plan for the query. The *physical planner* transforms the logical plan into the physical plan, selecting suitable implementations where applicable. For example, a scan may decide to use an existing index instead of scanning the base tables based on selectivity estimates, or switch between a hash join or merge join depending on the join predicates.

DuckDB uses a vectorized interpreted *execution engine* [9]. This approach was chosen over Just-in-Time compilation (JIT) of SQL queries [58] for portability reasons. JIT engines depend on massive compiler libraries (e.g. LLVM) with additional transitive dependencies. DuckDB uses vectors of a fixed maximum amount of values (1024 by default). Fixed-length types such as integers are stored as native arrays. Variable-length values such as strings are represented as a native array of pointers into a separate string heap. `NULL` values are represented using a separate bit vector. This allows fast intersection of `NULL` vectors for binary vector operations and avoids redundant computation. To avoid excessive shifting of data within the vectors when

e.g. the data is filtered, the vectors may have a selection vector, which is a list of offsets into the vector stating which indices of the vector are relevant [9]. DuckDB contains an extensive library of vector operations that support the relational operators, this library expands code for all supported data types using C++ code templates.

The execution engine executes the query in a so-called *"Vector Volcano"* model. Query execution commences by pulling the first "chunk" of data from the root node of the physical plan. A chunk is a horizontal subset of a result set, query intermediate or base table. This node will recursively pull chunks from child nodes, eventually arriving at a scan operator which produces chunks by reading from the persistent tables. This continues until the chunk arriving at the root is empty, at which point the query is completed.

DuckDB provides ACID-compliance through Multi-Version Concurrency Control (MVCC). We implement HyPer's serializable variant of MVCC that is tailored specifically for hybrid OLAP/OLTP systems [60]. This variant updates data in-place immediately, and keeps previous states stored in a separate undo buffer for concurrent transactions and aborts. MVCC was chosen over simpler schemes such as Optimistic Concurrency Control because, even though DuckDB's main use case is analytics, modifying tables in parallel was still an often-requested feature in the past.

For persistent storage, DuckDB uses a custom single-file storage layout inspired by the DataBlocks layout [50]. The single-file is partitioned into separate fixed-size blocks that hold the data of individual columns. When the columns are too small to fill up a single block, multiple columns can be packed together into a block to avoid wasting space. The table metadata lives in separate blocks and carries lightweight indexes for the individual blocks that the table points to, including min/max indices for every block that allow for the skipping of reading certain blocks into memory.

# 3 Summary

In this chapter, we have presented the embedded analytical database system DuckDB. DuckDB is a purpose-built embedded analytical database system that offers efficient

execution of analytical workloads and a very fast interface between the database system and analytical tools. It is built to be embedded and solves many of the problems faced by MonetDBLite with regards to resource usage and robustness. It is easy to setup and install with zero external dependencies and can be installed through standard package and library managers of popular analytical tools.

# CHAPTER 8

# Conclusion

## 1 Big Picture

In this thesis, we have investigated each of the methods in which analytical tools can be combined with relational database management systems. For each of the methods, we have provided improvements in both the efficiency and the usability departments.

Each of the three methods that we have considered has its place. As for our original goal of making the RDBMS as easy to use as flat file storage; the embedded database systems definitely shine. They are easy to install, and once installed can be used directly from within the analytical tools with very little setup overhead.

However, they share one of the same drawbacks as working with flat files in that they are not designed for collaborating with multiple people over multiple machines. In these scenarios, setting up a separate database server is the preferred solution.

The client-server connection is effective when the user wants to export the data and run an analysis pipeline only once. In this scenario, our proposed client-server protocol can significantly accelerate the speed of data export and assist the user in running their analysis pipeline efficiently.

When the user wants to run their analysis pipelines several times, for example as part of reporting software that periodically generates new graphs based on new data, MonetDB/Python UDFs shine as the stand-alone server architecture can be combined with the fast data transfer between the RDBMS and the analytical tool.

# 2 Future Research

In this section, we will present potential future research directions in the area of combining analytical tools and RDBMSs. We split up this section by each of the different methods of combining analytical tools with RDBMSs and discuss future research directions for each of the different methods.

## 2.1 Client-Server Connections

### Adaptive Compression

In our current protocol, we use a simple heuristic to determine which compression method to use. An optimization that can be made to our protocol is therefore to use the network speed as a heuristic for which compression method to use. Using compression methods that offer degrees of compression, the cost of the compression can be fine tuned and dynamically adapted to changing network conditions.

### Parallel Serialization

Further performance could be gained over our proposed protocol by serializing the result set in parallel. This can be advantageous for parallelizable queries. In these scenarios, the threads can immediately start serializing the result set to thread-local buffers as they compute the result without having to wait for the entire query to finish. However, since writing the data to the socket still has to happen in a serialized fashion we only expect performance gains in a limited set of scenarios. For example, when result set serialization is expensive due to heavy compression or when the query is fully parallelizable.

## 2.2 In-Database Processing

### Polymorphism

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

The problem with polymorphic return types is that the return types of the function must be known while constructing the query plan in the current execution engine. Thus we cannot execute the function and look at the returned values to determine the column types. The solution proposed by Friedman et al. [30] is to allow the user to create a function that specifies the output columns of the function based on the types of input columns. This function is then called while constructing the query plan to determine the output types of the function.

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

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

### Data Partitioning

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

However, as noted by Jaedicke et al. [44], 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 2-1a. The current partitioning scheme does not rearrange the values in the columns, which allows these identifiers to remain virtual. However, if we rearrange the values in the columns to match a user-defined partitioning scheme, we would need to explicitly store the row identifiers, resulting in significant additional overhead. This is avoided by the special partitioning used for computing parallel aggregates, because we do not need to know the individual tuple identifiers of each of the values as we are accumulating the actual values, thus we only need to know the group that the value belongs to.

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

### Distributed Execution

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

### Query Flow Optimization

Currently, we treat MonetDB/Python functions 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, estimations could be made.

It has been suggested by Hellerstein et al. [38] and Chaudhuri et al. [13] 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. [19] 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.

### Script Optimization

In this thesis we have focused mainly on optimizing the dataflow around user-defined functions. We have seen in Figure 4-5 that this dramatically speeds up functions for which transportation of data is the main bottleneck. However, when the computation time dominates the transportation time this optimization will not provide a significant speedup. We have provided the ability to execute functions in parallel, which can still

provide significant speedups to these functions. However, we still treat the user-defined functions as black boxes. Additional speedups could be achieved by looking into the user-defined functions and optimizing the code within the functions.

### Cardinality Estimation

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

### Code Translation

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

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

## 2.3 Embedded Databases

### Embeddability

In MonetDBLite, there are still several open issues that result from the nature of how MonetDBLite was created. Because the database that is based on, MonetDB, operates as a stand-alone server several limitations are present in the code that introduce problems when it is used as an embedded database.

MonetDB traditionally only allows a single database process to read the same database. There are is no fine grained locking between several database processes. Instead, a global lock is used on the entire database. If the user attempts to start a database server with a database that is currently occupied by another server an error will be thrown ("database locked") and the process will exit. This makes sense in the stand-alone server scenario, as running multiple database servers on the same database does not make much sense. However, it is a problem in the embedded database scenario because multiple processes might want to access the same database.

Another limitation is that the MonetDB server can only run on a single database at a time because of the large amount of global variables present in the codebase of MonetDB. This is no problem in the stand-alone server case, because another server can be started in a different database directory. However, for the embedded case, this is a limitation because only a single database can be opened in the same process.

As MonetDB is designed to run on a large machine that is dedicated to running the database server and has enough memory to handle the working set, MonetDB also does not perform gracious handling of out-of-memory situations. In many places in the codebase, `malloc` returning a `NULL` is not handled and will lead to the database server crashing. The processing model of MonetDB also does not lend itself well to low memory devices, as large intermediates are materialized entirely in memory. In the case of smaller devices, this may lead to the system frequently swapping or even running out of disk space and crashing the server.

These issues are so ingrained into the MonetDB codebase that they are very difficult to address. In fact, fixing these issues will require almost a complete rewrite of the entire codebase. Instead, we have decided to write DuckDB from scratch in order to fix these issues.

### Hardware Distrust

Hardware distrust is another unexplored area of database research that is very relevant to embedded database systems. As the embedded database runs on low quality consumer hardware instead of high quality server hardware it is very possible that the

system is ran on broken hardware. In the case of broken hardware, it is important that the database system prevents (or at the very least limits) the corruption of the data stored by the database system.

Any component used by the database system can be broken and can be broken in different ways. The hard disk can report successful writes even when writes have not occured, or it can flip bits within the file. Random bits can be flipped in the memory, or entire memory regions can be corrupt. Even the CPU can return incorrect results when broken or overclocked. Detecting and attempting to limit the damage caused by broken hardware components without significantly impacting performance is an area that we are actively working on in DuckDB.

### Resource Contention

Embedded database systems always run alongside their host application, and the resources used by the host application can vary wildly. The host application can be either a simple shell that performs almost no additional work or a full-fledged analytical application that consumes large amounts of memory and CPU resources.

Currently DuckDB employs the standard solution of letting the user configure the resource consumption of the database system. While this works, it adds additional knobs for the user and does not allow for adaptive resource balancing of the database system. For example, the database system could switch to using less memory as the host system requires more memory, or switch to using more CPU resources when these resources become available.

# Bibliography

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating Compression and Execution in Column-oriented Database Systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 671–682, New York, NY, USA, 2006. ACM.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[3] 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*, pages 287–290. AAAI Press, 1996.

[4] Grant Allen and Mike Owens. *The Definitive Guide to SQLite*. Apress, Berkely, CA, USA, 2nd edition, 2010.

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, et al. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International*

*Conference on Management of Data*, SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[6] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System r: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, June 1976.

[7] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A fast dynamic language for technical computing. *CoRR*, abs/1209.5145, 2012.

[8] Peter Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *In CIDR*, 2005.

[9] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005*, pages 225–237, 2005.

[10] United States Census Bureau. American Community Survey. Technical report, 2014.

[11] Pierre Carbonnelle. Top IDE index. 2018.

[12] D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured english query language. In *ACM SIGMOD*, page 249264, 5 1974.

[13] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, June 1999.

[14] Qiming Chen, Meichun Hsu, and Rui Liu. Extend UDF Technology for Integrated Analytics. In TorbenBach Pedersen, MukeshK. Mohania, and AMin Tjoa, editors, *Data Warehousing and Knowledge Discovery*, volume 5691 of *Lecture Notes in Computer Science*, pages 256–270. Springer Berlin Heidelberg, 2009.

[15] Qiming Chen, Meichun Hsu, Rui Liu, and Weihong Wang. Scaling-up and speeding-up video analytics inside database engine. In SouravS. Bhowmick, Josef Kng, and Roland Wagner, editors, *Database and Expert Systems Applications*, volume 5690 of *Lecture Notes in Computer Science*, pages 244–254. Springer Berlin Heidelberg, 2009.

[16] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[17] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M Hellerstein, and Caleb Welton. MAD skills: new analysis practices for big data. *Proceedings of the VLDB Endowment*, 2(2):1481–1492, 2009.

[18] Yann Collet. LZ4 - Extremely fast compression. Technical report, 2013.

[19] Andrew Crotty, Alex Galakatos, Kayhan Dursun, Tim Kraska, Carsten Binnig, Ugur Cetintemel, and Stan Zdonik. An architecture for compiling udf-centric workflows. *Proc. VLDB Endow.*, 8(12):1466–1477, August 2015.

[20] Anthony Damico. American Community Survey (ACS). Technical report, 2018.

[21] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[22] Matt Dowle and Arun Srinivasan. *data.table: Extension of 'data.frame'*, 2019. R package version 1.12.0.

[23] Dwayne Richard Hipp. Create Or Redefine SQL Functions. In *SQLite User Manual*.

[24] Jon Ellis and Linda Ho. JDBC 3.0 Specification. Technical report, Sun Microsystems, October 2001.

[25] Greenplum Database 4.2. Technical report, EMC Corporation, 2012.

[26] Xixuan Feng, Arun Kumar, Benjamin Recht, and Christopher Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 325–336, New York, NY, USA, 2012. ACM.

[27] Lukas Fittl. C library for accessing the postgresql parser outside of the server environment. `https://github.com//fittl/libpg_query`, 2019.

[28] Free Software Foundation. The GNU C Library - Memory-mapped I/O. Technical report, Free Software Foundation, 2018.

[29] The GNU C Library - Memory Protection. Technical report, Free Software Foundation, 2018.

[30] Eric Friedman, Peter Pawlowski, and John Cieslewicz. Sql/mapreduce: A practical approach to self-describing, polymorphic, and parallelizable user-defined functions. *Proc. VLDB Endow.*, 2(2):1402–1413, August 2009.

[31] Jean Gailly. gzip: The data compression program. Technical report, University of Utah, July 1993.

[32] Kyle Geiger. *Inside ODBC*. Microsoft Press, 1995.

[33] Chris Gibson, Kris Katterjohn, Mixter, and Fyodor. Ncat Reference Guide. Technical report, Nmap project, 2016.

[34] Protocol Buffers: Developer's Guide. Technical report, Google, 2016.

[35] Anurag Gupta, Deepak Agarwal, Derek Tan, et al. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1917–1923, New York, NY, USA, 2015. ACM.

[36] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng,

Kun Li, and Arun Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, August 2012.

[37] Joseph M. Hellerstein, Christoper R, U. Wisconsin, Aleksander Gorajek, Kun Li, U. Florida, Kee Siong Ng, U. Wisconsin, Caleb Welton, Daisy Zhe Wang, U. Florida, Xixuan Feng, and U. Wisconsin. The MADlib analytics library, or MAD skills, the SQL.

[38] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD Rec.*, 22(2):267–276, June 1993.

[39] Stephen Hemminger. Network Emulation with NetEm. Technical report, Open Source Development Lab, April 2005.

[40] Pedro Holanda, Mark Raasveldt, and Martin Kersten. Don't Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes. In *Proceedings of the 28th International Conference on Simpósio Brasileiro de Banco de Dados, SSBD 2017, Uberlndia, Brazil*, 2017.

[41] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull*, 2012.

[42] MongoDB Inc. MongoDB Architecture Guide. Technical report, MongoDB Inc., June 2016.

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

[44] 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, pages 379–389, New York, NY, USA, 1998. ACM.

[45] Michael Jaedicke and Bernhard Mitschang. User-defined table operators: Enhancing extensibility for ordbms. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick

Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 494–505. Morgan Kaufmann, 1999.

[46] Steinar H. Gunderson Jeff Dean, Sanjay Ghemawat. Snappy, a fast compressor/decompressor. Technical report, Google, 2016.

[47] Roger Magoulas John King. 2015 data science salary survey. September 2015.

[48] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2015-10-01].

[49] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, et al. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.*, 5(12):1790–1801, August 2012.

[50] Harald Lang, Tobias Mühlbauer, Florian Funke, et al. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 311–326, 2016.

[51] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *CoRR*, abs/1209.2137, 2012.

[52] Volker Linnemann, Klaus Küspert, Peter Dadam, Peter Pistor, R. Erbe, Alfons Kemper, Norbert Südkamp, Georg Walch, and Mechtild Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proceedings of the 14th International Conference on Very Large Data Bases*, VLDB '88, pages 294–305, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[53] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development.* " O'Reilly Media, Inc.", 2012.

[54] Thomas Lumley. Package 'survey'. 2018.

[55] John C McCallum. Memory Prices (1957-2019). Technical report, April 2019.

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

[57] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 539–552, 2008.

[58] Thomas Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[59] Thomas Neumann and Alfons Kemper. Unnesting arbitrary queries. In *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 383–402, 2015.

[60] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. Fast serializable multi-version concurrency control for main-memory database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 677–689, 2015.

[61] Thomas Neumann and Bernhard Radke. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 677–692, New York, NY, USA, 2018. ACM.

[62] Department of Transport Statistics. Airline On-Time Statistics and Delay Causes. Technical report, United States Department of Transportation, 2016.

[63] Carlos Ordonez and Sasi K Pitchaimalai. One-pass data mining algorithms in a DBMS with UDFs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1217–1220. ACM, 2011.

[64] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[65] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[66] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

[67] PostgreSQL Development Team. Procedural Languages. In *PostgreSQL User Manual*.

[68] Andrew Prunicki. Apache Thrift. Technical report, Object Computing, Inc., June 2009.

[69] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.

[70] Mark Raasveldt and Hannes Mühleisen. Don't Hold My Data Hostage: A Case for Client Protocol Redesign. *Proc. VLDB Endow.*, 10(10):1022–1033, June 2017.

[71] Bert Rich. *Oracle Database Reference, 12c Release 1*. 2017.

[72] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[73] 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, pages 343–354, New York, NY, USA, 1998. ACM.

[74] Felix Martin Schuhknecht, Jens Dittrich, and Ankur Sharma. RUMA Has It: Rewired User-space Memory Access is Possible! *Proc. VLDB Endow.*, 9(10):768–779, June 2016.

[75] Lefteris Sidirourgos and Martin Kersten. Column imprints: A secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 893–904, New York, NY, USA, 2013. ACM.

[76] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next Generation Database Management System. *Commun. ACM*, 34(10):78–92, October 1991.

[77] Michael Stonebraker and Greg Kemnitz. The POSTGRES Next Generation Database Management System. *Commun. ACM*, 34(10):78–92, October 1991.

[78] Carl-Fredrik Sundlöf. In-database computations. Master's thesis, Royal Institute of Technology, Sweden, 10 2010.

[79] MySQL Development Team. Adding a New User-Defined Function. In *MySQL User Manual*.

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

[81] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, et al. Hive - a petabyte scale data warehouse using Hadoop. *2014 IEEE 30th International Conference on Data Engineering*, 0:996–1005, 2010.

[82] Transaction Processing Performance Council. TPC Benchmark H (Decision Support) Standard Specification. Technical report, Transaction Processing Performance Council, June 2013.

[83] Jan Urbaski. Postgres on the wire. In *PGCon 2016*, May 2014.

[84] S. van der Walt, S.C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science Engineering*, 13(2):22–30, March 2011.

[85] Manasi Vartak, Harihar Subramanyam, Wei-En Lee, Srinidhi Viswanathan, Saadiyah Husnoo, Samuel Madden, and Matei Zaharia. Model DB: a system

for machine learning model management. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*, page 14. ACM, 2016.

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

[87] Hadley Wickham. Package 'DBI'. 2018.

[88] Hadley Wickham, Romain François, Lionel Henry, and Kirill Müller. *dplyr: A Grammar of Data Manipulation*, 2018. R package version 0.7.8.

[89] Michael Widenius and Davis Axmark. *MySQL Reference Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2002.

[90] Florian Wolf, Iraklis Psaroudakis, Norman May, Anastasia Ailamaki, and Kai-Uwe Sattler. Extending database task schedulers for multi-threaded application code. In *Proceedings of the 27th International Conference on Scientific and Statistical Database Management*, SSDBM '15, pages 25:1–25:12, New York, NY, USA, 2015. ACM.

[91] Paul C. Zikopoulos and Roman B. Melnyk. *DB2: The Complete Reference*. McGraw-Hill Companies, January 2001.

# Summary

The database research community has made tremendous strides in developing powerful database engines that allow for efficient analytical query processing. However, these powerful systems have gone largely unused by analysts and data scientists. This poor adoption is caused primarily by the state of database-client integration: current methods of combining databases with analytical tools are slow and cumbersome. Instead, data scientists have opted to re-invent database systems by developing a zoo of data management alternatives that perform similar tasks to classical database management systems, but have many of the problems that were solved in the database field decades ago.

In this thesis we attempt to overcome this challenge by investigating how we can facilitate efficient and painless integration of analytical tools and relational database management systems. We focus our investigation on the three primary methods for database-client integration: client-server connections, in-database processing and embedding the database inside the client application.

For each of these methods we take an extensive look at implementations in existing systems, and evaluate how they perform in the context of standard analytical workloads. We evaluate the benefits and drawbacks that they exhibit in this context, both in terms of query performance and usability.

We propose several novel techniques that improve upon the state-of-the-art. We demonstrate a new client-server protocol that is optimized for bulk-transfer of large data sets. We showcase our MonetDB/Python UDFs, that improve on large in-database processing efficiency through vectorized execution. We describe MonetDBLite, an embedded version of the MonetDB database system that we have efficiently integrated with Python and R. The techniques that we propose have all been integrated and tested in real database systems, showing that these solutions are not just theoretical but practically applicable as well.

In the final chapter we showcase DuckDB, a new data management system that we have built from scratch. When building DuckDB, we took all the lessons that we learned from developing efficient database-client interfaces and applied them.

In conclusion, the techniques that we have developed enable significantly more efficient and usable integration between database systems and analytical tools. Nevertheless, there is still more to be explored in this area. We close this thesis with a program for future research, as well as sketches for solutions to them.

# Samenvatting

Database onderzoekers hebben enorme voortgang geboekt in het ontwikkelen van krachtige database systemen die efficient analytische queries kunnen beantwoorden. Deze krachtige systemen worden echter zelden gebruikt door analytici. Dat komt voornamelijk omdat het gebruik van huidige relationele database systemen in combinatie met de programma's die zij gebruiken traag en onhandig is. In plaats van deze database systemen te gebruiken, zijn analytici database systemen opnieuw aan het uitvinden. Ze schrijven hun eigen programma's die vergelijkbare functionaliteit hebben,maar de innovaties van het database veld van de afgelopen decennia negeren.

In dit proefschrift proberen we dit probleem op te lossen. We doen dit door te onderzoeken hoe we de integratie van database systemen met deze analytische programma's efficinter en gebruiksvriendelijker kunnen maken. Ons onderzoek is gefocussed op de drie primaire methodes van database-client integratie: client-server verbindingen, in-database analyses en gentegreerde database systemen.

Voor elk van deze methoden onderzoeken wij de implementaties in bestaande database systemen, en evalueren wij hoe efficient deze zijn voor standaard analytische gebruik. We kijken naar de voor en nadelen van elk van deze technieken, zowel in termen van efficiencie als in gebruiksvriendelijkheid.

We introduceren meerdere nieuwe technieken die verbeteren op de huidige state-of-

the-art. We demonstreren een nieuw client-server protocol dat wij hebben ontwikkeld dat geoptimalizeerd is voor bulkoverdracht van grote data sets. We laten onze MonetDB/Python user-defined functions zien, die efficiente grootschalige in-database analyses versnellen door gebruik te maken van vectorisatie. Uiteindelijk beschrijven wij MonetDBLite, een versie van het MonetDB database systeem die wij hebben gentegreerd in R en Python. Al onze technieken zijn getest in de context van echte systemen, wat laat zien dat onze oplossingen niet alleen theoretisch maar ook praktisch toepasbaar zijn.

In het laatste hoofdstuk introduceren wij DuckDB, een nieuw data management systeem dat wij hebben gebouwd met als specifiek doel om deze analytici te ondersteunen. Bij het bouwen van DuckDB hebben wij alle lessen die we hebben geleerd over de integratie van database systemen met analytische applicaties toegepast.

In conclusie, de algoritmes die wij hebben ontwikkeld maken het mogelijk om database systemen veel efficienter te integreren met analytische applicaties. Desalniettemin is er nog meer om te onderzoeken in dit gebied. We sluiten dit proefschrift af met suggesties voor toekomstig onderzoek, samen met ideen voor eventuele oplossingen.

# Publications

This thesis is based on the following set of publications:

- **Vectorized UDFs in Column-Stores**, Mark Raasveldt, Hannes Mühleisen, 28th International Conference on Scientific and Statistical Database Management (SSDBM 2016)
- **Don't Hold My Data Hostage - A Case For Client Protocol Redesign**, Mark Raasveldt, Hannes Mühleisen, 43rd International Conference on Very Large Data Bases (VLDB 2017)
- **Dont Hold My UDFs Hostage - Exporting UDFs For Debugging Purposes**, Mark Raasveldt, Pedro Holanda and Stefan Manegold, 32nd Simpósio Brasileiro de Bancos de Dados (SBBD 2017)
- **Deep Integration of Machine Learning Into Column Stores**, Mark Raasveldt, Pedro Holanda, Hannes Mühleisen and Stefan Manegold, 21st International Conference on Extending Database Technology (EDBT 2018)
- **MonetDBLite: An Embedded Analytical Database**, Mark Raasveldt and Hannes Mühleisen (Unpublished)
- **devUDF: Increasing UDF development efficiency through IDE Integration. It works like a PyCharm!**, Mark Raasveldt, Pedro Holanda and Stefan Manegold, 22nd International Conference on Extending Database Technology (EDBT 2019, Demo Track)
- **DuckDB: an Embeddable Analytical Database**, Mark Raasveldt and Hannes Mühleisen, ACM International Conference on Management of Data (SIGMOD 2019, Demo Track)

# Curriculum Vitae

Mark Raasveldt geboren op 20 September 1992 te Leiderdorp

| | |
|---|---|
| 2016 - 2020 | PhD candidate<br>Database Architectures group<br>Centrum van Wiskunde & Informatica (CWI)<br>Supervised by Hannes Mühleisen and Stefan Manegold |
| 2013 - 2015 | Master of Science<br>Computing Science (Cum Laude)<br>Utrecht University |
| 2010 - 2013 | Bachelor of Science<br>Computer Science<br>Utrecht University |
| 2004 - 2010 | High School<br>Tweetalig VWO, Profiel N/T<br>Scala College |