Integrating Analytics with Relational Databases

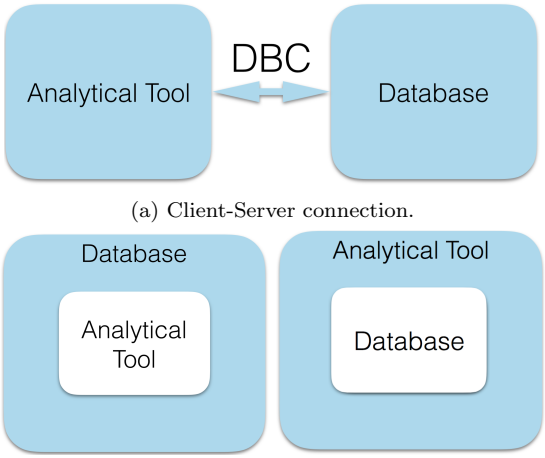
Mark Raasveldt
supervised by Hannes Mühleisen and Stefan Manegold
Centrum Wiskunde & Informatica
Amsterdam, Netherlands
m.raasveldt@cwi.nl

ABSTRACT

In order to uncover insights and trends, it is an increasingly common practice for companies of all shapes and sizes to gather large quantities of data and to then analyze that data. This data can come from a multitude of different sources, ranging from data gathered about consumer behavior to data gathered from sensors. The most prevalent way of storing and managing data has traditionally been a relational database management system (RDBMS). However, there is currently a disconnect between the tools used for analysis of data and the tools used for storing that data. Instead of working directly with RDBMSes, these tools are build to work in a stand-alone fashion, and offer integration with RDBMSes as an afterthought. The focus of my PhD research is on investigating different methods of combining popular analytical tools (such as R or Python) with database management systems in an efficient and user-friendly fashion.

1. INTRODUCTION

There is a disconnect between data-intensive analytical tools and traditional database management systems. Data scientists using these tools often prefer to manually manage their data by storing it either as structured text (such as CSV or XML files), or as binary files [8]. This approach of managing data introduces a lot of problems, especially when a large amount of data from different sources has to be managed or combined. Flat file storage requires tremendous manual effort to maintain, is often difficult to reason about because of the lack of a rigid schema and is difficult to share between multiple users. Furthermore, modifying the data is prone to corruption because of lack of transactional guarantees and atomic write actions. Another consequence of this disconnect is that data scientists have re-implemented many common database operations inside libraries such as dplyr [16] or Pandas [9]. Instead of performing joins or aggregations using a RDBMS, they perform them using these libraries. However, these libraries suffer from having to load all required data and intermediates into memory, leading to frequent out of memory problems or poor performance due to swapping.



(a) Client-Server connection.

(b) In-database processing. (c) Embedded database.

Figure 1: Different ways of connecting analytical tools with a database management system.

These issues could be solved through the use of a RDBMS. The RDBMS can prevent data corruption through ACID properties, it can automatically manage data storage for the user and make data easier to reason about by enforcing a rigid schema. In addition, the RDBMS can perform efficient execution on larger-than-memory data, and allows concurrent read and write access to the data in a safe way.

Popular analytical tools such as R or Python can be used in conjunction with database systems. There are SQLite bindings for these languages, and it is possible to connect to a database server using standard client connector protocols. However, data scientists prefer to use flat file storage methods over these existing approaches, because these connections are either inefficient or inconvenient to use.

The focus of this work is on identifying the problems encountered when combining a RDBMS with analytical tools, and on implementing various solutions to overcome these issues to allow for both a more efficient and more flexible combination of these tools. Figure 1 shows the three main methods in which a relational database can be combined with an analytical tool. We investigate each of these methods, and attempt to improve them from both a usability and a performance perspective.

Proceedings of the VLDB 2018 PhD Workshop, August 27, 2018. Rio de Janeiro, Brazil.

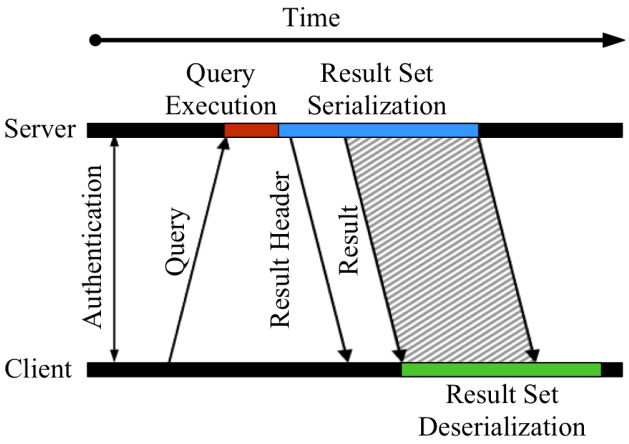


Figure 2: Communication between a client and server.

2. 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 1a. In this way, 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 can issue queries to the database, after which the server will compute the answer to the query and transfer the results to the client through the socket. This process is shown in Figure 2.

In order to perform analytics on data stored inside the database, the data is exported from the database to the analytical tool over the socket connection, after which that data is processed in the client. The main advantage to this approach is that it is mostly database agnostic, as the standardized ODBC or JDBC connectors can be used to connect to almost every database. In addition, it is relatively easy to integrate into existing pipelines as the loading from flat files can be replaced by loading from a database without having to touch the rest of the pipeline.

However, this approach is problematic when dealing with a large amount of data as is often required in modern analytical pipelines. 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' memory.

In our work [13], we perform a survey of popular RDBMS and note that they are not optimized for the scenario of high-volume data export. They take a significant amount of time to export a relatively small amount of data even when the server and client are located on the same machine or connected through a high-throughput network connection. This is because existing client protocols were designed for the transfer of a small amount of rows in OLTP workloads, and have significant per-tuple and per-value overheads that result in the slow export of large tables.

To remedy this problem, we investigate the different design choices that can be made when designing a result set serialization format, and we propose a new client protocol that is optimized for the transfer of large amounts of data from the server to the client. By using a column-major chunk-wise format that utilizes lightweight compression and a binary format that is close to the native database format, we can export large tables an order of magnitude faster than existing solutions.

However, even with a client protocol optimized for this scenario, there is still a significant amount of time required to push data "over the wire". In addition, as this approach only replaces the loading of data from a flat file storage system with the loading of data from the RDBMS into the client, it still requires the entire dataset and intermediates to fit inside the clients' main memory.

3. 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 1b.

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* [7]. However, this small number of functions and aggregates is not sufficient to perform complex data analysis tasks [15].

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. [4]. 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 [3].

In order to make it easier to perform in-database analytics, we introduced MonetDB/Python UDFs [12] in the Open-Source DBMS MonetDB [6]. These user-defined functions can be written in Python, and process code in a vectorized way. The input and output variables of the functions and aggregates can be provided as either standardized NumPy arrays [14] or Pandas DataFrames [9]. In this way, the user-defined functions mimic the execution of regular analytical Python programs and can be written without any knowledge of the database internals. Because of their vectorized nature, the heavy interpreter overhead is not incurred once for every tuple but only once for every invocation of the function. Combined with the use of zero-copy techniques for both the input and output columns these functions can be executed efficiently on large datasets.

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

Listing 1: Chain of SQL operators.

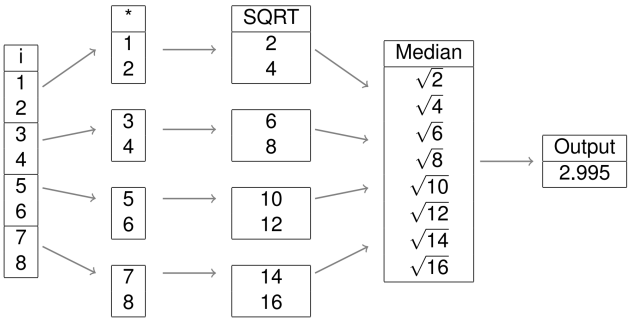


Figure 3: Parallel operator chain of Listing 1.

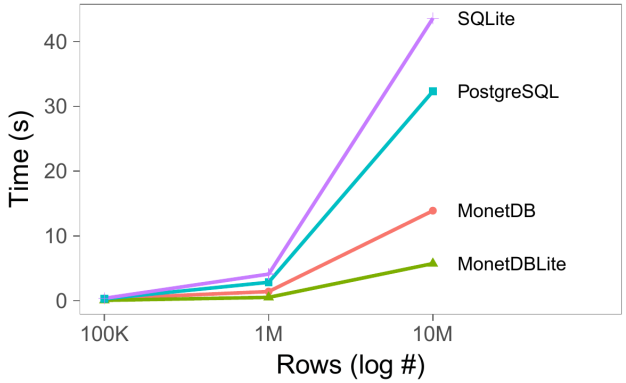


Figure 4: Transfer of the TPC-H lineitem table from the database to the client process.

Parallelism. Another advantage of UDFs is that they can take advantage of the databases' automatic parallelization model. In MonetDB, parallel execution is achieved by marking individual operators as either parallelizable or blocking. When a chain of parallel operators is executed on a column, the column is split up into several chunks and the operator is executed once on each chunk. When a blocking operator is encountered, the chunks are packed into a single column and the blocking operator is executed on that column. This process is visualized in Figure 3.

MonetDB/Python UDFs can be parallelized in the same way. The functions can be set to either allow parallelization, in which case they are executed as a parallelizable operator, or to disallow parallelization, in which case they will operate with the entire column as input. User-defined aggregates are parallelized over the different groups, where the aggregate is called once for each group with the tuples belonging to that group as input. The aggregates computed for each group are then gathered and combined to form the final result.

Development Workflow. A challenge when developing user-defined functions is that, since they are executed inside the database server, standard tools and integrated development environments (IDEs) cannot be used to develop them. As a result, developers cannot use sophisticated debugging techniques (e.g., Interactive Debugging) and have to resort to inefficient debugging strategies to make their code work.

In order to make it easier to develop MonetDB/Python UDFs, we extended the client of MonetDB to allow for local testing of user-defined functions [5]. The required data (or a sample of it) is automatically shipped from the database to the client together with the source code of the UDF. It can then be executed locally and run in either a stand-alone interactive debugger or a full-fledged IDE.

Model Management. Another issue that arises is the management of different machine learning models. Current systems, such as TensorFlow, allow the models to be written to disk as individual files. However, much like handling data as flat files, handling models as flat files is cumbersome and error-prone.

In our work [11], we investigate how we can do model management using a relational database. By storing the models in a relational database, we can store the models alongside their training information or meta-information gathered about the model. This allows us to query and apply the models based on this information, as well as apply multiple models in parallel for ensemble learning.

4. EMBEDDED DATABASE

Both the previously managed approaches require the user to have 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 a database inside the client program, as shown in Figure 1c, 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 full-fledged database systems.

The most commonly used embedded database is SQLite [2]. However, SQLite is first and foremost designed for transactional workloads. It is a row-store database that uses a volcano-based processing model for query execution. While popular analytical tools such as Python and R do have SQLite bindings, it does not perform well when used for analytical purposes. Even exclusively using SQLite as a storage engine typically does not work out well in these scenarios. Often only select columns of a table are used in analyses, and its row-wise storage layout forces it to always load entire tables. This can lead to very poor performance when dealing with wide data.

To fill this gap, we created MonetDBLite [10], an Open-Source embedded database based on the popular columnar database MonetDB. Much like SQLite, it is an in-process database that can be installed and run directly from within popular analytical tools without any external dependencies. However, unlike SQLite it is designed for analytical workloads, and as such performs significantly better when executing analytical queries that operate on large amounts of data. Because of the columnar layout of the database and zero-copy semantics, data can be copied between the database and the analytical tool for a constant cost, and no large costs need to be paid when extracting only a subset of columns from a wide table.

This efficient data transfer is illustrated in the experiment in Figure 4, where we transfer the `lineitem` table from the TPC-H benchmark [1] from the database to the client process using MonetDBLite, SQLite, MonetDB and PostgreSQL. We observe that data can be exported from MonetDBLite an order of magnitude faster than over either a

socket connection (in the case of MonetDB and PostgreSQL) or from the row-storage model of SQLite.

5. RESEARCH DIRECTIONS

While our current solutions have made it both easier and more efficient to combine relational databases with analytical tools, connecting them efficiently and effortlessly is by no means a solved problem. In this section, we will describe the open research problems that we have identified and how we plan on tackling them in the future.

Automatic Code Shipping. While user-defined functions allow for efficient in-database analytics, it still requires significant manual transformation effort to take an existing analytical pipeline and make it run inside the database server. Ideally, we would be able to automatically translate an existing analytical pipeline and execute it on data residing in the database without requiring manual user effort.

A solution to this problem could be to take a program that uses a database connector to connect to a database, and run the code directly inside the database server. Instead of connecting with the database through a socket, the SQL code could be directly executed inside the server and the results could be used inside the analytical tool without requiring data transfer. This approach does negate the potential advantages of automatic parallelization, however.

Alternatively, the code could be analyzed and translated into user-defined functions that can be executed within the database server and could potentially be parallelized. Analyzing if arbitrary code could be safely parallelized is not possible, though, as it would be equivalent to solving the Halting problem. However, it would already be useful if a limited subset of operations could be automatically shipped and executed in parallel inside the database server. For example, a number of commonly used operations of the Pandas and NumPy libraries could be supported.

UDF Co-optimization. Currently, MonetDB/Python UDFs are executed as black-box functions. As a result, there is almost no room for automatic optimization of the actual code. The only optimization we apply is the parallelization of the functions, however, even this requires the user to tell us whether or not the function is parallelizable.

Lazy evaluation could allow us to optimize the UDFs more. Rather than executing the function in an eager manner, we could defer the execution of certain operations on the input columns (e.g. common NumPy and Pandas operations). This would allow us to build a computation graph, and either (1) run parts of that computation graph inside the databases' execution engine (where it could be executed in parallel and take advantage of existing indexes), or (2) feed information extracted from the computation graph to the database optimizer.

Acknowledgments

This work was funded by the Netherlands Organisation for Scientific Research (NWO), project "Process Mining for Multi-Objective Online Control".

6. REFERENCES

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

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

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

[4] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The MADlib Analytics Library: Or MAD Skills, the SQL. *Proc. VLDB Endow.*, 5(12):1700–1711, Aug. 2012.

[5] P. Holanda, M. Raasveldt, and M. 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.

[6] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull*, 2012.

[7] ISO. ISO/IEC 9075:1992, Database Language SQL. Technical report, International Organization for Standardization (ISO), July 1992.

[8] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2917–2926, Dec. 2012.

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

[10] M. Raasveldt. MonetDBLite: An Embedded Analytical Database. *SIGMOD '18: Proceedings of the 2018 ACM International Conference on Management of Data*, 2018.

[11] M. Raasveldt, P. Holanda, H. Mühleisen, and S. Manegold. Deep Integration of Machine Learning Into Column Stores. In *Proceedings of the 21st International Conference on Extending Database Technology (EDBT)*, 2018.

[12] M. Raasveldt and H. Mühleisen. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management, SSDBM 2016, Budapest, Hungary, July 18-20, 2016*, pages 16:1–16:12, 2016.

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

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

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

[16] H. Wickham. Package 'dplyr': A Grammar of Data Manipulation, 2017.