Summary Paper of *Don't Hold My Data Hostage – A Case For Client Protocol Redesign*

Authors of Original Paper: Mark Raasveldt and Hannes Mühleisen

Hannes Pohnke
pohnke@tu-berlin.de
TU Berlin
Berlin, Germany

ABSTRACT

Traditionally, database query processing and ML tasks are executed on separate, dedicated systems, but the current trend goes towards integrated data analysis pipelines that combine both tasks. In state of the art systems, orchestration of those two tasks still is inefficient due to expensive data transfer and missed global optimization potential. The paper we are summarizing, "Don't Hold My Data Hostage – A Case For Client Protocol Redesign" (DHMDH) by Mark Raasveldt and Hannes Muhleisen, addressed this problem by investigating the high cost of transferring large data from databases to the client programs, which can be much more time consuming than the actual query execution. The authors explored and analysed serialization methods, that were used in database systems and identified their inefficiencies through various experiments. They also introduced a new columnar serialization method that significantly enhanced data transfer performance. By improving the data transfer, this approach was a step towards efficiently combining database and machine learning systems.

KEYWORDS

Databases, Client Protocols, Data Export

ACM Reference Format:

Hannes Pohnke. 2018. Summary Paper of *Don't Hold My Data Hostage – A Case For Client Protocol Redesign*: Authors of Original Paper: Mark Raasveldt and Hannes Mühleisen. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation emai (Conference acronym 'XX)*. ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Back in the day popular Machine Learning (ML) tools, including RapidMiner, Weka, R's ML packages, and Python-based toolkits like SciKit-Learn and TensorFlow, only supported bulk data transfer from databases [?]. Therefore, users had to manually load data that was already in a table format into these tools. The problem is that the transfer of such data is very slow, even when both database and ML workloads are located on the same system. To avoid transfer overheads, this often resulted in smaller data samples being used, which is generally a bad thing in ML.

When the DHMDH paper was introduced in 2017 most research in this field, including the authors' [?], focused on performing computations within the database, avoiding the need for data transfer. Since then a lot of approaches for optimizing the transfer arose, which can be broadly categorized in server-side and client-side optimizations [?]. Raasveldt and Mühlheisen chose a client-side approach, as they realized that the biggest and easiest to implement optimization potential lay in addressing the overhead of result set serialization (RSS). RSS refers to converting data into a format suitable for transfer. The summarized paper examined state of the art serialization formats and explored the design space of those formats in client protocol design. Key contributions include:

- Benchmarking current RSS methods, to identify data transfer inefficiencies.
- Investigating techniques for creating efficient serialization methods.
- Proposing a new column-based serialization method with significant performance improvements.

2 STATE OF THE ART

All remote database systems use client protocols to manage communication between server and client. This process begins with an initial authentication and an exchange of meta data. Then the client can send queries to the server. Once the server processes the query, it has to serialize the query data into a result set format, send it over the socket to the client, and then the client deserializes the data to use it. As pointed out, the time spent on these steps is significantly influenced by the design of the result set format. We will now dive into the exploration and evaluation of serialization formats used by leading systems in 2017, that were conducted by the authors.

2.1 Overview

To evaluate the performance of various databases for large result set exports, experiments on systems like MySQL [36], PostgreSQL [32], IBM DB2 [37], "DBMS X", MonetDB [5], Hive [33], and MongoDB [23] were conducted. MySQL's client protocol with GZIP compression ("MySQL+C") was tested separately. Of course there are more database systems, but many of those adopt client protocols from more popular databases to reuse existing implementations. This

Conference acronym 'XX, June 03–05, 2018, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

selection of systems therefore seems representative for the state of the art in 2017.

The experiments focused on isolating the duration for RSS and data transfer. The TPC-H benchmark's SF10 lineitem table was loaded into each database, and data retrieval times were measured using ODBC connectors (JDBC for Hive). Netcat [12] was used as a baseline for efficient data transfer without any database overheads. The results we can see in Table 1, showed that transferring data as CSV via Netcat was drastically faster than using any tested database system. This undermined the problem of slow RSS and transfer in general. MongoDB had the highest overhead due to its document-based style, while MySQL+C transferred the least data due to compression.

Table 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

2.2 Network Impact

Those experiment were conducted with both server and client located on the same machine, so the data transfer time wasn't influenced by network factors. However, network limitations can significantly impact the performance of different client protocols. Low bandwidth makes transferring bytes more costly, favoring compression and smaller protocols, while high latency increases the cost of sending confirmation packets. To simulate network limitations, the netem[17] utility was used to create scenarios with restricted bandwidth (0.1 ,1 , 10, 100 ms) and increased latency (10, 100, 1000 Mb/s), and 1 million rows of the lineitem table were transferred. The chosen scenarios seem representative, because they capture the typical range those parameters take in real life, from home network to internet transfers.

Higher latency adds a fixed cost to sending messages, affecting connection establishment but not large result set transfers, at least that was the assumption. Contrary to that, high latency negatively impacted all systems, due to TCP/IP layer acknowledgements, which occur frequently and slow down data transfer.

Reduced throughput increases the cost of sending messages based on their size, penalizing protocols that send more data. With lower throughput, protocols that normally perform well, like PostgreSQL or MongoDB, suffer significantly. However, systems that use compression like MySQL+C perform better than the others as the main bottleneck shifts to data transfer, making the cost of (de)compression less significant

2.3 Result Set Serialization

To understand the differences in time and bytes transferred across various database protocols, their data serialization formats were examined. The authors did that by looking at the hexadecimal representations of a tiny sample table of the different protocols. This highlights the actual data versus overhead in the data representations.

PostgreSQL

- Format: Each row is transferred in a separate message, including total length, number of fields, and field-specific data lengths.
- Overhead: High per-row metadata and redundant information, leading to more bytes transferred.
- Efficiency: Low (de)serialization costs, resulting in quick transfers if network conditions aren't limiting.

MySQL

- Format: Uses binary encoding for metadata and text for field data. Rows begin with data length, followed by a sequence number and length-prefixed field data.
- Overhead: Sequence numbers are redundant due to TCP guarantees; variable-length integers for field lengths.
- Efficiency: Efficient binary encoding for metadata but larger data size due to text representation of field data.

DBMS X

- Format: Terse protocol with each row prefixed by a packet header and values preceded by length in bytes.
- Overhead: Uses variable-length integers for lengths; employs a configurable fixed message length for batch transfers.
- Efficiency: Computationally intensive but allows performance optimization through configuration.

MonetDB

- Format: Text-based serialization transferring ASCII representations of values. Rows are delimited like CSV, with additional formatting characters.
- Overhead: Formatting characters inflate size; conversion between binary and string formats is costly.
- Efficiency: Simple format but expensive in terms of computational resources due to text conversion.

Hive

- Format: Thrift-based columnar format with serialization for structured messages, including meta data for reassembly.
- Overhead: Verbose serialization with significant space wasted on NULL mask encoding.
- Efficiency: Poor performance in benchmarks due to expensive variable-length encoding for integer columns, despite the columnar format.

3 PROTOCOL DESIGN SPACE

When choosing a protocol design, there is always a core trade-off: computational versus transfer costs. For instance, when computation is negligible, using heavyweight compression techniques like XZ [31] can substantially trim transfer expenses. Conversely, if transfer costs are not a problem, opting for reduced computational overhead, even at the expense of increased data transfer, can speed up the protocol. To benchmark all the design choices, they were isolated and tested on 3 datasets:

- **SF10 lineitem**, resembling real-world data warehouses, 16 columns, 60 million rows, no missing values, 7.2GB
- **American Community Survey (ACS)** data, 274 columns, 9.1 million rows, 16.1% missing values, 7.0GB [6].
- **Airline On-Time Statistics**, 109 columns, 10 million rows, 55.2% missing values, 3.6GB [25].

The objective was to uncover how the following design choices shape serialization format performance.

Row/Column-wise. When designing data transfer protocols, there is a fundamental choice between serializing data row-wise or column-wise. Most systems opt for row-wise serialization, aligning with popular database APIs like ODBC and JDBC. However, columnar formats can offer significant advantages, because data stored in a column-wise format compresses much more effectively [1]. Additionally, many data analysis tools, such as R [29] and the Pandas [22], already store data column-based. Using a row-based protocol introduces an unnecessary overhead in those cases. Despite its advantages, a pure column-based format requires an entire column to be transferred before moving to the next. If a client requires row-wise data access, it must first read and cache the entire result set, which is impractical for large datasets. To address this while keeping the advantages of columnar formats, a vector-based protocol is proposed. In this method, chunks of rows are encoded in a column-based format. This allows the client to cache only the rows of a single chunk, rather than the entire result set.

Chunk Size. Choosing the right chunk size for data transfer is crucial. Larger chunks require more memory, while very small chunks lose the benefits of a columnar protocol. Testing different chunk sizes (2KB to 100MB) with three datasets, both uncompressed and with Snappy compression [14], revealed that small chunk sizes lead to bad perform and low compression ratios. This is because the model becomes more and more row-based up to the point of 2kb chunks where there might be only one row inside a chunk. Optimal performance and compression occurred at around 1MB chunks, indicating that efficient data transfer with a vector-based format doesn't need large memory allocations.

Data Compression. Data compression is vital for improving performance on limited network throughput. However, it involves trade-offs between the costs for compression and the achieved compression ratio (how much smaller is data after compression). Lightweight tools like Snappy [14] and LZ4 [7] prioritize fast compression but sacrifice ratio, while heavyweights like XZ [31] compress slowly but to a higher degree. GZIP [10] is a balanced approach. The experiment of sending the lineitem table through differently fast networks undermined, that the optimal compression method varies based on the cost of transferring bytes. Lightweight methods suit fast networks where extra bytes are cheap, while slower networks benefit from better ratios despite computational overhead. The problem with the heavyweight compressions is that they really only are better than the others for very slow networks (10Mbit/s) and even then, do not perform well overall. Lightweight compressions are better than no compressions for connections of a gigabit per second downwards. It is not feasible to change the compression method with every individual server-client connection speed, which is why the authors decided to go with a simple heuristic: no compression for local connections and lightweight compression for most realistic network scenarios, such as LAN or reasonably high-speed connections.

Column-Specific Compression. Column-specific compression methods, such as run-length encoding or delta encoding for numeric columns, should offer higher compression ratios at lower costs than generic compression algorithms, when applied on integer value columns. In experiments transferring only integer columns from the datasets though, binpacking and PFOR [21] (colmn-specific) did not consistently outperform Snappy [14] (generic). In fact they only performed well on lineitem. For ACS these methods performed poorly due to the amount of columns, which led to an overhead of processing small data chunks, and for ontime they struggled with the large value range within the columns. Therefore, the authors decided against the use of column-specific compression. The use of datasets with different characteristics was a great choice by the authors. This made sure their benchmarks are representative and unbiased. When using only one or similar datasets, it would have been easy to come to false conclusions on that aspect.

Data Serialization. Data serialization for TCP sockets can be achieved through custom or generic methods like Protocol Buffers [13] or Thrift [28]. The authors built a custom format, trying to make it more similar to the actual data storage layout, in order to reduce the computational overhead. They conducted experiments transferring the lineitem table and claimed that their custom format "significantly outperforms" Protobuf. This might be true for a local Network, but not for a WAN setting. As the WAN setting is the more common scenario this statement by the authors can be questioned, at least based on the shown data. Nevertheless, they opted for the custom serialization format.

String handling. Serializing character strings presents challenges to which there are three main solutions: Null-Termination (string ends with a 0 byte), Length-Prefixing (string is prefixed with its length), and Fixed Width (all strings have maximum string length of column). To evaluate these methods, the authors transferred different string columns from the lineitem table, using both uncompressed and Snappy-compressed protocols:

- **S**ingle-Character Column (l_returnflag): Fixed-width performed best. Length-prefixing and null-termination had twice the byte transfer of fixed width.
- **L**onger Column (l_comment): Fixed-width transferred much more bytes due to padding, leading to higher compression costs. The other methods had comparable performance.

Increasing VARCHAR widths made the fixed-width much worse, despite improved compressibility. Therefore, the authors decided to use fixed width only for a VARCHAR width of 1 and for larger strings they preferred null termination.

4 IMPLEMENTATION & RESULTS

Finally, we will have a look at the authors' client protocol and its performance against state of the art protocols from 2017.

4.1 MonetDB Implementation

In MonetDB, the protocol serializes query results into column major chunks, where each chunk has a prefix with the row count. Fixed

Table 2: Results of transferring the SF10 lineitem table for different network configurations.

	Timings (s)			
System	**TLocal**	**TLAN**	**TWAN**	**Size (GB)**
(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

length columns have no extra prefix, while variable length columns are prefixed with their total byte length for efficient row-wise access. Missing values use special values for their specific domain. During authentication, the client sets the maximum chunk size in bytes. This ensures that every chunk fits in a buffer, except for very wide rows. The server adjusts the rows per chunk to stay within this limit and if there are rows that are not fitting, it notifies the client to increase the buffer. By having every chunk inside one buffer, the client can access this data without any unnecessary conversion or copying. The data is copied into the buffer in column-wise order or compressed directly into it, depending on the setting. The chunk is then sent to the client, again with the possibility to compress it beforehand. Setting row counts per chunk is normally constant, except for BLOB or CLOB columns, where a scan determines the rows per chunk due to variable row sizes.

4.2 PostgreSQL Implementation

A big difference to the Monet DB implementation is the handling of missing values: each column is prefixed with a bitmask indicating missing values. Because of this bitmask, fixed length columns from before are now also variable in length, which is why every column is prefixed with its length in the PostgreSQL serialization. This allows the client to skip columns without scanning the data and prevents data transfer for empty rows.

The other difference is that the authors had to transform the data from PostgreSQL's inherent row-based format into the more efficient column-based format. A challenge when doing so were the null masks, which made it impossible to know the row size in advance. Therefore, data for each column is first copied to a temporary buffer when iterating over the rows and when its full copied to the stream buffer. The potentially inefficient access pattern for converting row-major to column-major format is mitigated by keeping chunks small enough to fit in the L3 cache of a CPU.

4.3 Evaluation

To assess the real-world performance of their protocols, the authors compared them with the state-of-the-art client protocols from chapter 2, using the datasets we introduced in chapter 3. Again, they benchmark this additionally with the transfer of a simple CSV using netcat [12] with different compressions. The experiments were conducted on a Linux VM with 16GB memory and 8 CPU cores. Both the database and the client were running inside this VM. As in chapter 2.2, the netem [17] utility was used to simulate different network scenarios: Local, LAN (1000 Mb/s throughput, 0.3ms latency) and WAN (100 Mbit/s throughput, 25ms latency, 1

5 CONCLUSION & RELATED WORK

asd

5.1 Related Work

Fazit nochmal labern was impact ist und wie sich anscheinend in database solutions durchsetzen aus mehreren argumenten wenn das tatsächlich der fall ist? kp ich glaube sogar nicht so wie sich connector x anhört

6 REFERENCES