Integrating analytics with relational databases

Raasveldt, M.

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

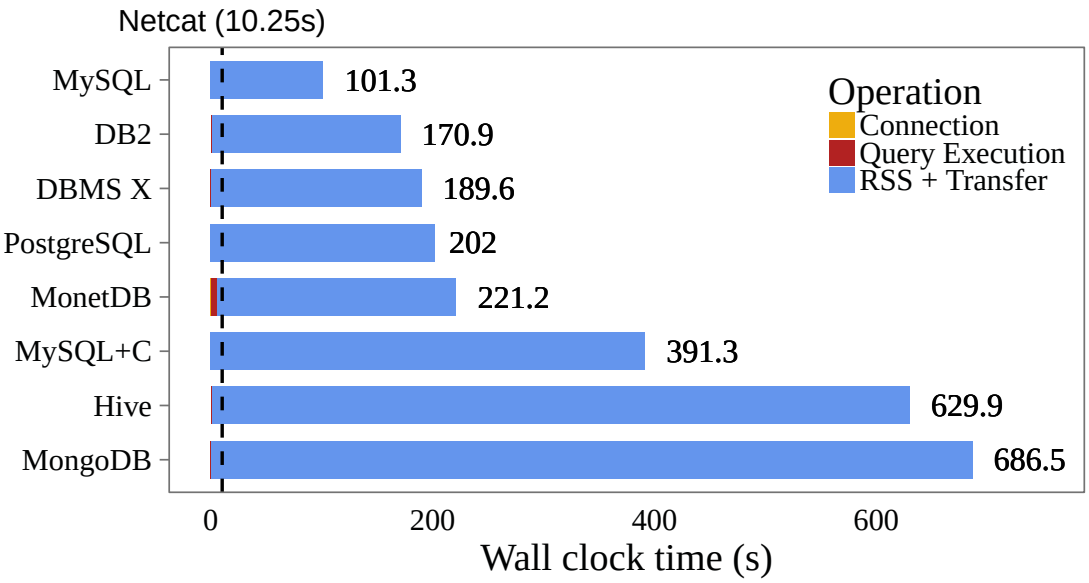

Figure 3-1: Wall clock time for retrieving the `lineitem` table (SF10) over a loopback connection. The dashed line is the wall clock time for netcat to transfer a CSV of the data.

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.

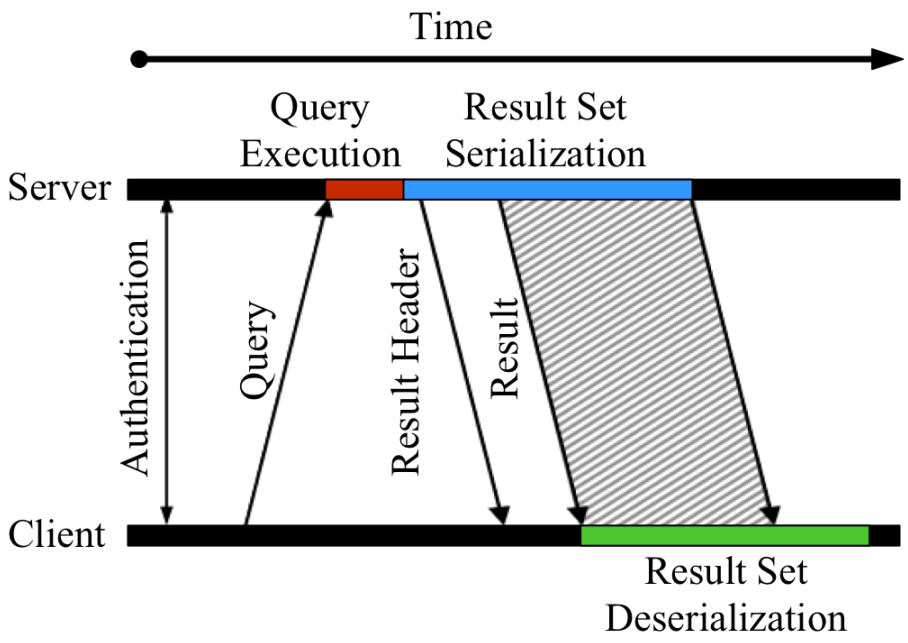

Figure 3-2: Communication between a client and a server

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.

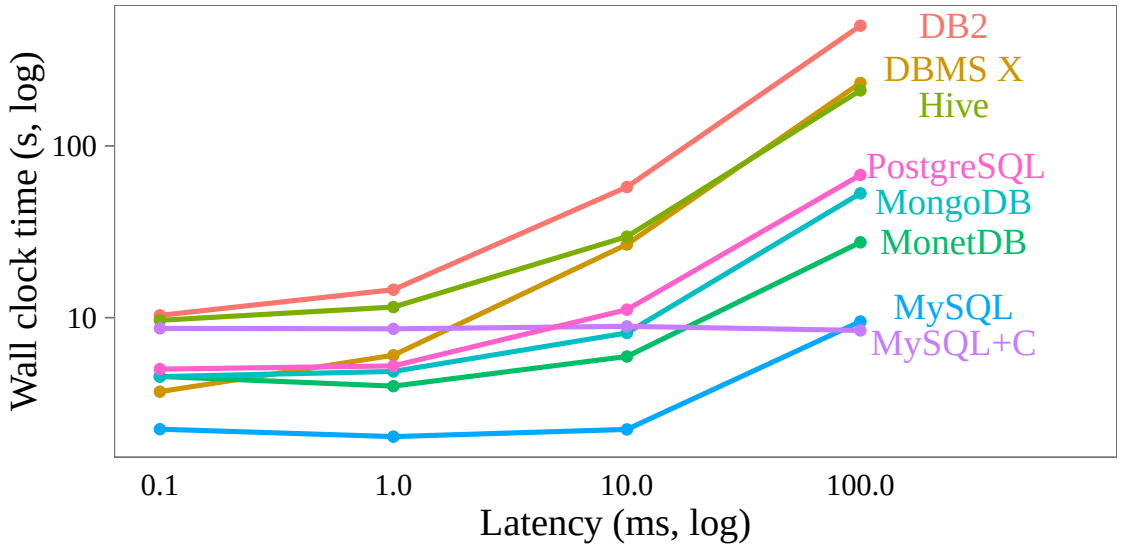

Figure 3-3: Time taken to transfer a result set with varying latency.

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

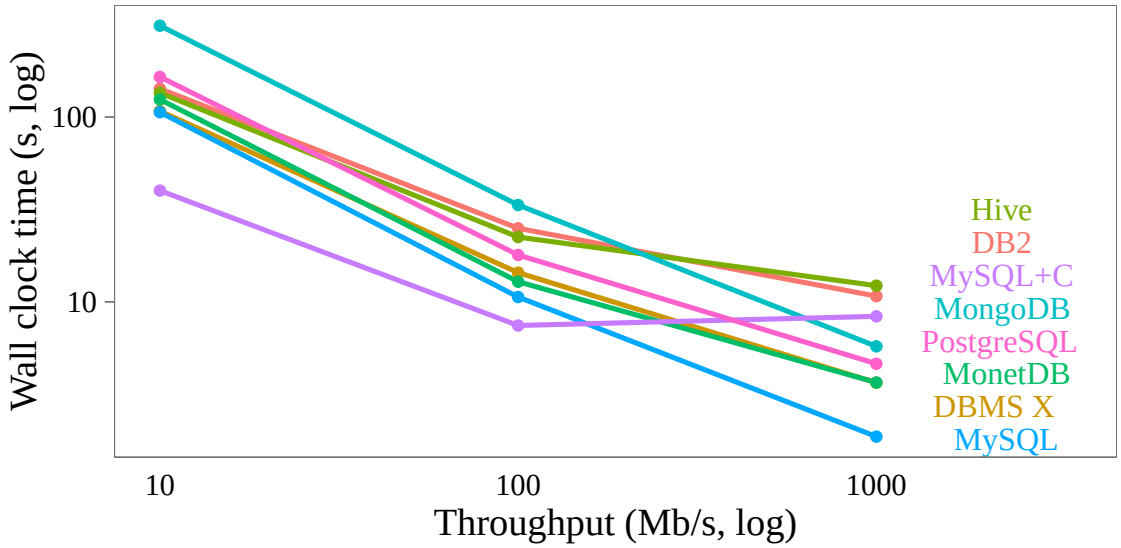

Figure 3-4: Time taken to transfer a result set with varying throughput limitations.

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.

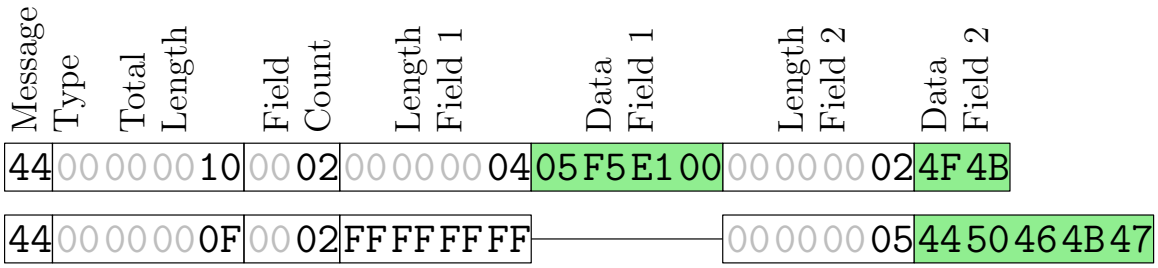

Figure 3-5: PostgreSQL result set wire format

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.

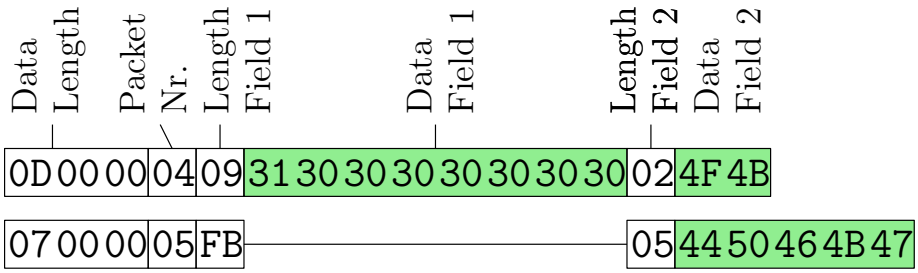

Figure 3-6: MySQL text result set wire format

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.

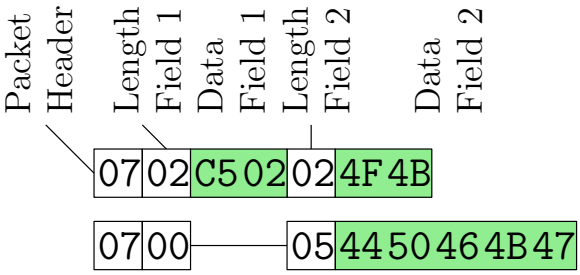

Figure 3-7: DBMS X result set wire format

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

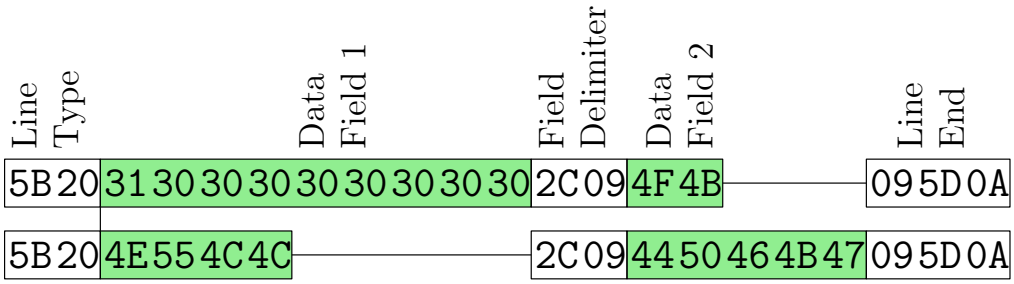

Figure 3-8: MonetDB result set wire format

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.

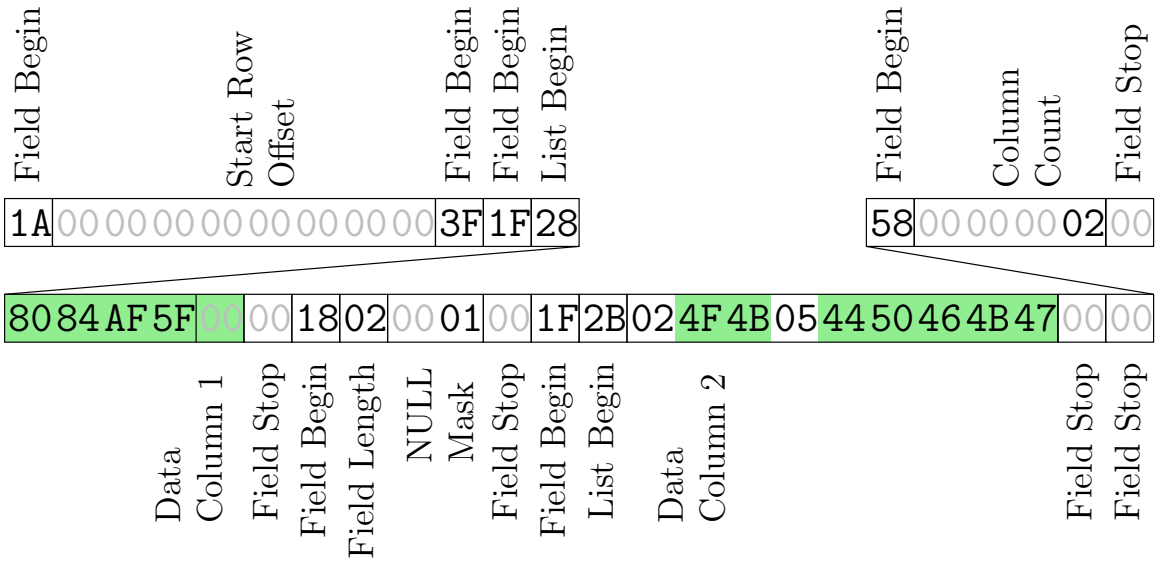

Figure 3-9: Hive result set wire format using "compact" Thrift encoding

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	1.4×10^1	55.9	6.56	1.38
	10KB	7.1×10^1	15.2	5.92	1.80
	100KB	7.1×10^2	10.9	5.81	2.12
	1MB	7.1×10^3	**10.0**	**5.80**	2.25
	10MB	7.1×10^4	10.9	**5.80**	**2.26**
	100MB	7.1×10^5	13.3	6.15	2.23
ACS	2KB	1.0×10^0	281.1	11.36	2.06
	10KB	8.0×10^0	46.7	9.72	3.18
	100KB	8.5×10^1	16.2	9.50	3.68
	1MB	8.5×10^2	**11.9**	**9.49**	3.81
	10MB	8.5×10^3	15.3	9.50	**3.86**
	100MB	8.5×10^4	17.9	10.05	3.84
Ontime	2KB	1.0×10^0	162.9	8.70	2.13
	10KB	8.0×10^0	27.3	4.10	4.15
	100KB	8.5×10^1	7.6	3.47	8.15
	1MB	8.6×10^2	6.9	3.42	9.80
	10MB	8.6×10^3	**6.2**	**3.42**	10.24
	100MB	8.6×10^4	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.

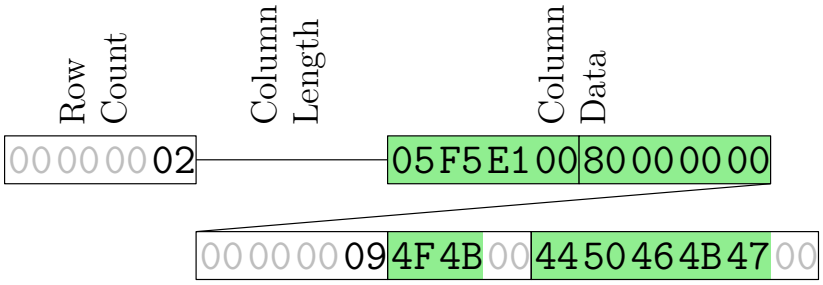

Figure 3-10: Proposed result set wire format – MonetDB

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.

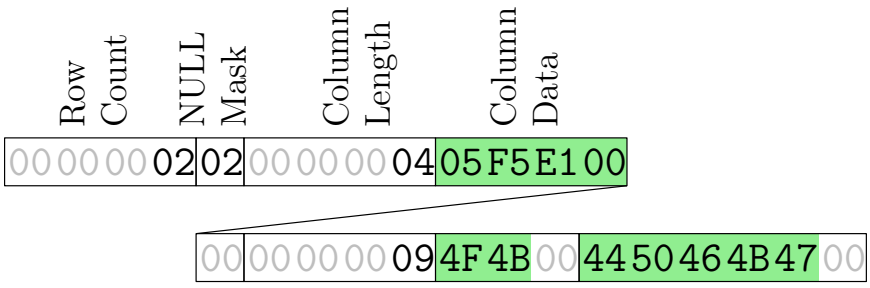

Figure 3-11: Proposed result set wire format – PostgreSQL

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.