Technische Universität Berlin

Chair of Database Systems and Information Management

Bachelor's Thesis

Efficient and Generic Data Movement for Decentralized Query Execution Environments

Joel Ziegler
Degree Program: Computer Science
Matriculation Number: 371459

Reviewers
Prof. Dr. Volker Markl
Prof. Dr. Odej Kao

Advisor
Haralampos Gavriilidis

Submission Date
07.10.2022

Hereby I declare that I wrote this thesis myself with the help of no more than the mentioned literature and auxiliary means.

Berlin, 30.09.2022

. . . . . .

Firstname, Lastname(s)

Abstract

Data movement between systems is a crucial factor in today's data infrastructure, among other things it is indispensable in the extract-transform-load processes of many data analytics use cases. Additionally, while the classic work flow gathers data on a single system or mediator and performs analytics local, the potential of decentralized query execution, harnessing specialized processing advantages of different database systems, gets recognized.

This thesis analyzes data migration pipelines, their performance bottlenecks and their implementation costs and uses that knowledge to propose an architecture for generic and efficient data movement, called Warpdrive. Warpdrive is fully compatible with any database system and lays the groundwork for decentralized query execution, where systems transfer data between each other without a mediator.

We provide a prototype implementation of the Warpdrive, its data transfer libraries and data wrappers to connect PostgreSQL and Clickhouse instances. We show, that even in its early prototype stage, Warpdrive is already comparable to generic and specialized state-of-the-art data transfer methods.

Zusammenfassung

Die Bewegung von Daten zwischen Systemen ist ein entscheidender Faktor in der heutigen Dateninfrastruktur. Sie ist unter anderem für die Extrahier-Transformier-Lade-Prozesse vieler Anwendungsfälle der Datenanalyse unerlässlich. Während der klassische Arbeitsablauf Daten auf einem einzigen System oder Mediator sammelt und Analysen lokal durchführt, wird mehr und mehr das Potenzial einer dezentralisierten Abfrageausführung, die die spezialisierten Verarbeitungsvorteile verschiedener Datenbanksysteme nutzt, erkannt.

Diese Arbeit analysiert Datenmigrationspipelines, ihre Leistungsengpässe und ihre Implementierungskosten und nutzt dieses Wissen, um eine Architektur für generische und effiziente Datenmigration, genannt Warpdrive, vorzuschlagen. Warpdrive ist vollständig kompatibel mit jedem Datenbanksystem und bildet die Grundlage für eine dezentralisierte Abfrageausführung, bei der Systeme ohne Mediator Daten untereinander austauschen.

Wir stellen eine prototypische Implementierung von Warpdrive und seinen Datenübertragungsbibliotheken und Datenwrappern zur Verbindung von PostgreSQL- und Clickhouse-Instanzen vor. Wir zeigen, dass Warpdrive bereits in seinem frühen Prototypenstadium mit generischen und spezialisierten State-of-the-Art-Datentransfermethoden vergleichbar ist.

Acknowledgments

Many thanks to Haralampos Gavriilidis for being a great bachelor thesis mentor, supporting me with a lot of good advice and also knowing how to motivate undergraduates.

Additional thanks to Laurenz Albe for reliably providing fast and good PostgreSQL support at Stackoverflow (and other forums on the Internet) and Sutou Kouhei for providing help with Apache Arrow, even if the prototypes using Arrow did not make it into the final thesis.

Contents

1 Introduction

Data analytics, even though not a new field of science, continues to permeate into every part of our economy. For example, the continued development of new sensor types created new possibilities for data analytics use cases in the oil and gas industry [Mohammadpoor and Torabi(2020)]. The development of these new sources of data lead to an exponential growth in the amount of data [Manyika et al.(2011)], which increases the possibilities for data scientists to derive useful information, and the heterogeneity of data types, which complicates the analytic process. To accommodate for this heterogeneous data and the different use cases data scientists have for this data, database engineers developed a plethora of database management systems with different computational specializations, from relational database systems in row and column format to NoSQL database systems. As data analytics use cases commonly incorporate data from multiple sources, data movement between these sources is an integral part of the data analytics work flow. Still, there is a lack of data transfer methods, which are fast and readily available for all sources. Fast solutions require special connections between two systems, which are expensive to implement, while broadly compatible solutions exhibit poor performance.

The naive data transfer solution with the most compatibility is the use of CSV files. Their simplicity and long history ensured compatibility with nearly every database system in use. But as compatible as the CSV format is, it has obvious drawbacks in its String encoding of values, materialization of files and often manual managing of the file data transfer, which hinder its performance.

The SQL/MED standard targets the requirement for standardized access to external data by providing an interface for so called foreign data wrappers. While the standardized interface lays the groundwork for data access between any database system, current data wrapper designs still exhibit the disadvantage of compatibility vs performance. Generic wrapper implementations utilize JDBC or file transfer and do not exhibit adequate performance. Specialized solutions have poor compatibility and are not readily available. In addition, custom solutions incur an immense implementation overhead. BigDAWG itself, a modern polystore system, laments about the difficulty of implementing specialized data wrappers, which lead to months of development to include one additional DBMS into the polystore [Yu et al.(2019)].

This thesis aims to free the untapped specialized processing power of database

management systems for data analytics by aiming for the sweet-spot between efficient and fast data transfer and generic application of the transfer mechanism. We utilize SQL/MED as standard for foreign data access. For efficiency, we focused on fast data transfer independent of the network environment and allowed features like remote direct memory access and asynchronous sending operations. We also had to reduce serialization costs to a minimum, as serialization is a major contributor to slow data transfer between different database systems. To achieve generality, we planned our architecture and interfaces respectively and decoupled the data formats from the participating systems in the serialization process from each other. All in all, we are providing an architecture, called Warpdrive, a prototype library realizing the Warpdrive and experiments, comparing our prototypes with state-of-the-art foreign data wrappers. We show, that Warpdrive can compete with current solutions even in its early prototype state.

There are systems in development, which greatly benefit from a generic and efficient data transfer mechanism. Apache Wayang optimizes cross-platform data analytics as a framework over any kind of database system but does not concern itself with the data transfer itself. Poor transfer mechanisms in Apache Wayang would lead to poor cost estimations for nodes connected by these mechanisms. For some systems only basic transfer mechanisms exist. The Warpdrive could alleviate the potential for any database system to be connected and utilized. A more ambitious system is Agora, envisioned as a unified ecosystem for data analytics, bringing together data resources, computational resources and data analytics algorithms to allow any data scientist to access them and as a consequence greatly alleviates the potential benefit of data analytics as a whole. But to bring Agora into reality, many smaller inventions need to be incorporated, like Apache Wayang, and an efficient data transfer mechanism between database systems is an indispensable part of the system. Both also profit from the development in decentralized query execution, where database systems transfer data freely between each other and system specific computation advantages can be utilized. This thesis took care to be compatible and also alleviate decentralized query execution via SQL/MED compatibility and as generic data migration tool as a whole.

After introducing the necessary background in Section 2, we outline and analyze issues in current data movement techniques and argue why we need a hybrid solution in Section 3, introduce our Warpdrive architecture, which proposes to embed a client/server architecture into SQL/MED in Section 4 and evaluate Warpdrive prototype performance in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 Scientific Background

In this chapter, we give an overview of data movement between heterogeneous database management systems and describe our problem environment, explain the SQL/MED standard, which strongly influences our architecture and give an overview of state-of-the-art generic and specialized data transfer methods in decentralized query environments and their disadvantages we try to solve.

2.1 Data Movement in Heterogeneous DBMS Environments

Most database systems were not build with foreign data access and free data transfer in mind. To accommodate for modern use cases, which require increasingly more access to multiple heterogeneous database systems, bundles systems, called polystores, got developed. Polystores use handcrafted data transfer methods to allow access to every participating database system. Today, the Mediator/Wrapper architecture is the state-of-the-art data transfer method in most polystore systems [Bondiombouy and Valduriez(2016)]. The mediator allows querying multiple heterogeneous database systems with a unified querying language. Via wrappers, the mediator is able to query underlying database systems, collects the data and performs subsequent processing locally. While providing a solution to the desire of accessing data on heterogeneous systems, the problem of serializing and transferring different data formats is traded off by only having one system access every other participating system. This way, only one systems needs to be adjusted with specialized data transfer methods and the implementation overhead is reduced. Still, this architecture does not allow harnessing the processing power of participating systems for subqueries, generates a bottleneck in the mediator and the setup of the mediator is still costly. A new concept is that of a decentralized query environment, in which data can move freely between database systems and allow query execution of every participating database system to harness system specific computation advantages and reduce disadvantages of the Mediator/Wrapper architecture. This new concept is hindered by the complexity of data migration, which among other things encompasses the lack of a common data interface between database systems. The SQL/MED standard is an effort for such a common interface.

2.2 The SQL/MED Standard

SQL/MED stands for SQL Management of External Data [Melton et al.(2002)] and has been part of the SQL standard since 2003. SQL/MED extends SQL with syntax for accessing foreign databases via so called foreign tables, which get wrapped by a foreign data wrapper. The wrapper can wrap any external data source, be it SQL data or non-SQL, into a foreign table and abstracts the source from the SQL server. The SQL server queries the foreign table like any other table, the syntax abstracts the differences between foreign tables and local tables in queries, only the table creation hints at differences. SQL/MED allows any compliant SQL database to interface with foreign data sources via foreign data wrappers, while allowing the wrapper direct access to the internals of the database.

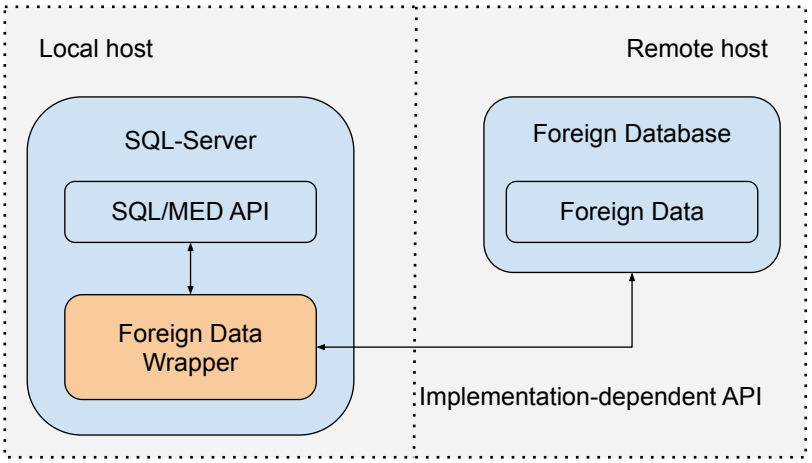


Figure 1: SQL-server access to a foreign database via SQL/MED. Adapted from SQL/MED status report [Melton et al.(2002)]. The foreign data wrapper can be third-party developed.

We aim to be compliant with SQL/MED foreign data wrappers, which currently are very implementation intensive to set up. Today, only a small subset of database systems are SQL/MED compliant, notably PostgreSQL and MariaDB, but we are confident, that compliance will rise, as SQL/MED is part of the SQL standard and the need for inter database data transfer will continue to rise.

2.3 The JDBC Interface

The generic data transfer solution with the most compatibility is the use of CSV files. Their simplicity and long history ensured compatibility with nearly every database system in use. But as compatible the CSV format is, it has obvious drawbacks in its String encoding of values, materialization of files and often manual managing of the file data transfer, which hinder its performance.

The generic transfer solutions JDBC and ODBC stem from a time, in which database systems mainly stored transactional data. There was a need for a generic access solution for client programs. Even though, JDBC and ODBC are different interfaces, we only describe JDBC, as they are similar in architecture and use case and share the same problems.

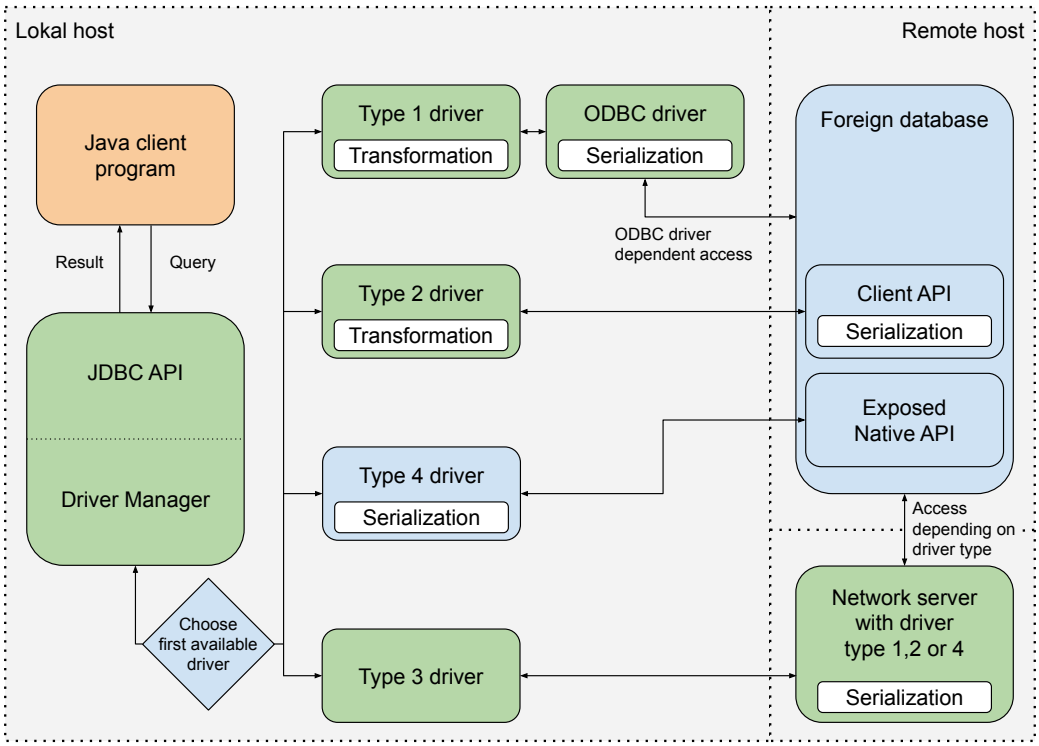


Figure 2: Data flow for foreign data access via JDBC. Green signals third party JDBC modules, orange signals developer provided modules and blue signals database system provided modules. Modeled after the JDBC Specifications 4.3 [Andersen(2017)].

The JDBC interface works in the following: A client program calls the JDBC API, which invokes the Driver Manager to access a desired database system via available JDBC drivers (see figure 2). The client now has the ability to query the foreign database and receive results in JDBCs ResultSet format. JDBC has a rich feature set for transactional workloads, but from the viewpoint of bulk data reading, major problems arise. Raasveldt et al. identify serialization costs from and into the JDBC ResultSet format and outdated transfer formats as the main performance bottlenecks in data transfer with JDBC [Raasveldt and Mühleisen(2017)]. Depending on the driver types available to the Driver Manager, multiple data transformations take place. The best available driver type is a type 4 driver, called a native driver. These JDBC drivers are implemented by the database system manufacturer themselves and offer the best achievable connection to a foreign database system. In the best case, type 4 drivers have direct access to the native data format of the foreign database system, serialize that format into the JDBC ResultSet from which the client program can read and optionally deserialize into its own format. Still the serialization into and from the JDBC ResultSet is cost intensive, as JDBC ResultSets are not optimized for bulk network transfer. Type 2 and Type 1 driver further increase the serialization problem, as they introduce additional intermediary formats into the data transfer and heavily rely upon the exposed interfaces of the foreign database. For example PostgreSQL exposes only a client API to access its data from outside the database system. To serve data to a client over the client API, PostgreSQL performs an extra serialization step to convert its internal data format into a client format. The client format is easier to handle for transactional workloads, but for data transfer to another database system, in which the goal is to convert retrieved data into the native format of the requesting system, this extra serialization step introduces unnecessary work into the data transfer and merely changes the format from which the type 2 driver serializes into the JDBC ResultSet format.

2.4 Specialized DBMS Connectors

Specialized data wrappers have the potential for the most efficient inter database data transfer, as they are implemented as one to one connections between two systems. Into this category, we count foreign data wrappers and direct modifications to database systems. Figure 3 shows the general scenarios of different specialized data transfer methods. One example of the first connection type can be found in Clickhouse servers. Clickhouse is a fast columnar database, which exposes its native format over a TCP connection. Multiple Clickhouse servers can access each others data via the native interface and do not require additional serialization steps. This is the most desired environment for data transfer, as serialization steps are

reduced to a minimum. Additionally this exposed native interface gives room for other types of specialized data transfer methods. Another database system could implement a data transfer, that queries Clickhouse data via its native format and only requires one transformation step to transform the Clickhouse data format into its local data format. If there is no exposed native interface in the foreign server, the last option for specialized data transfer methods is to access exposed client interfaces. This introduces an additional serialization step and resembles type 4 JDBC drivers. The difference is, the transfer format of type 4 drivers are in the native format of the foreign database system, which is an advantage compared to the client interface format, as native formats tend to be binary, therefore more space efficient and compressible, but then again serialize into the JDBC ResultSet, which may not be cheaper performance wise, than serializing into the client data format.

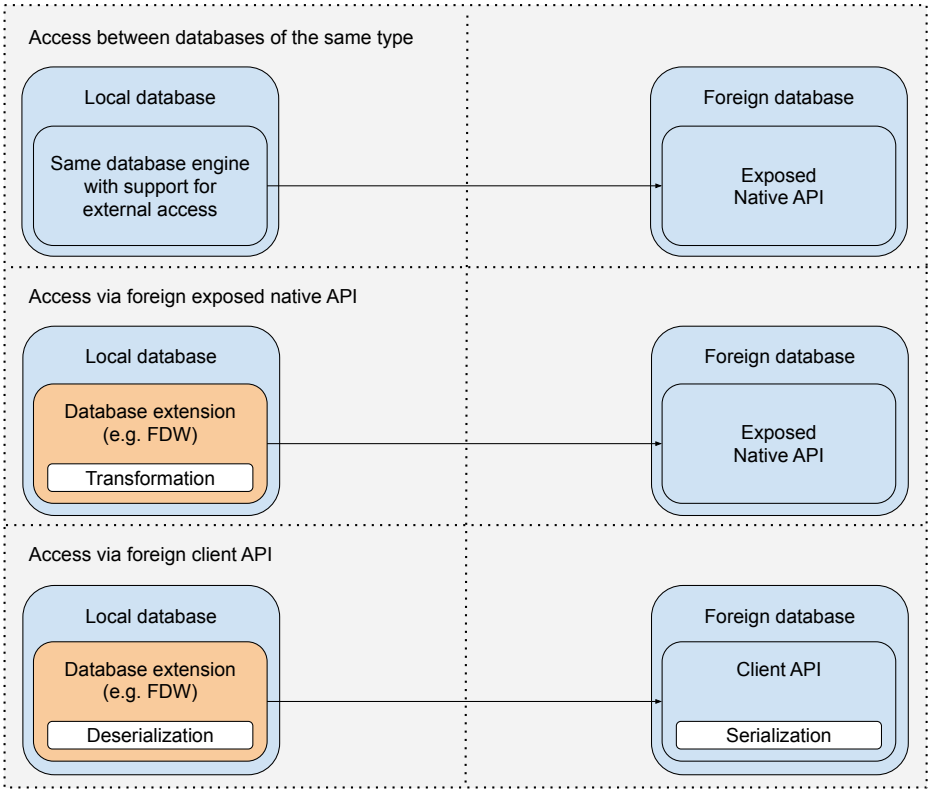


Figure 3: Architectures of specialized data wrappers. The number of serialization steps is based on the exposed interfaces of the participating database systems. Orange signals third party modules and blue signals database system provided modules.

So specialized data wrappers can come in three types, which are direct same database connections, one-time transformation connections between two native formats and two-time serialization connections between a native format and exposed non-native format interfaces. Their performance and capabilities vary greatly on the given features of participating databases and their implementation efforts scale directly to their performance benefits. The first type is only applicable for same database system connections and therefore would be realized by database system manufacturers. The second type is the ideal case specialized data wrapper.

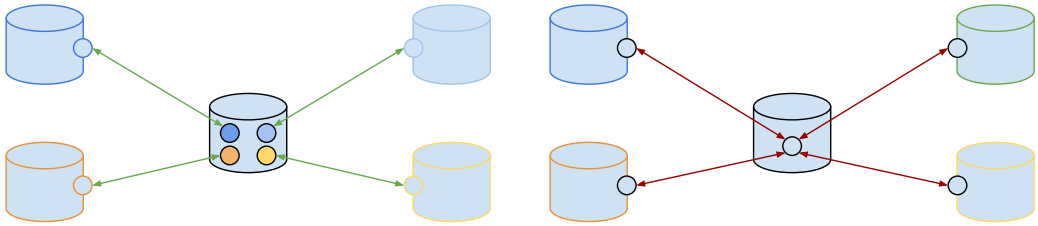
Even if database systems commonly expose their native formats and have internal extension capabilities, the problem of generality still troubles specialized data wrappers. Groups of database systems found in polystores or even database system networks, like envisioned in Agora, would need to implement a specialized data wrapper on every database system for every other database system type reachable, which amounts to exponential implementation overheads.

This problem also troubles BigDAWG, a polystore system by the Intel Science and Technology Center on Big Data [Gadepally et al.(2016)]. In their effort to develop a polystore system with internal data migration without a mediator/wrapper architecture and a single interface, they realized the immense implementation overhead needed, to connect further systems into the already established system. They write: "To add MySQL to the BigDAWG island took multiple months of effort." [Yu et al.(2019)]. BigDAWG only contained three other database systems at the time. One could imagine the effort needed to connect new database systems to an ever growing data federation. The problem of generality is as important as the problem of efficiency and we took care to address this problem in our architecture.

3 Issues in Generic and Specialized Data Movement

This thesis aim is to find the sweet spot of inter database data movement between generic, simple to set up transfer and specialized one to one transfer methods. In this chapter we lay out existing problems with inter-database data movements and derive requirements for our Warpdrive architecture.

The basic connection structure of a data federation through foreign data wrappers is seen in Figure 4. Wrappers can be categorized in specialized wrappers, only connecting two types of database systems with each other, and generic wrappers, connecting an arbitrary amount of database systems with each other. The difference between these wrappers came into the form of multiple serializations, unoptimized transfer formats and data transfer in generic wrappers and high implementation costs without the possibility of reusing the resulting implementation in specialized wrappers. In the following sections we will analyze these aspects of data transfer.



(a) Data Federation connected via specialized data wrappers

(b) Data Federation connected via generic data wrappers

Figure 4: Generic wrappers ease implementation, while impacting data transfer performance. Specialized wrappers on the other hand excel at transfer performance, but incur heavy implementation overheads.

Serialization Costs: Serializing in data migration means converting data into another format readable by the data destination, whether into an intermediate or the destination's native format. In itself, serialization is computation-intensive

as it encompasses a full pass over the data. Depending on the target format, serialization can further increase in cost, as redundancies are introduced or data encoding can be inconveniently bloated. For example, Raasveldt et al. found many redundancies and outdated information in database client data formats, which is not needed anymore in data transfer[Raasveldt and Mühleisen(2017)]. Therefore, one of our target goals is to avoid serialization, if possible.

Serialization can't be avoided in total, when designing a generic transfer solution, as the source and target database systems are different ones. Therefore, at least one serialization step is required. The trade off with the required serializations to be made is between implementation and serialization cost. If certain serialization steps are unavoidable, the question arises, how much implementation resources are used to ensure optimal serialization into native database formats. For example, data can be fed into a database system via client formats, which many systems expose. Our own experiences with implementing prototypes for PostgreSQL confirm, that it is easier implementation wise to serialize into client formats, than into native formats. This reduction of implementation cost comes with increased serialization cost in the form of unoptimized formats and further serialization steps, the database system itself has to take, to convert the given client format data into its native data.

Data Transfer: Data transfer in data migration is the task of sending bytes from a source to a destination machine. There are many kinds of data transfer today, depending on the connection types of the database systems machines, from plain transfer over sockets, over remote direct memory access with a minimum of data copying to shared memory space. Even though data transfer methods have been extensively researched, it is still a main concern for data migration to include fast transfer methods. Supporting a wide range of transfer methods can also be a substantial factor in implementation cost and has to be considered in designing our architecture.

The trade off here is between the data transfer performance and again the implementation cost. A simple and basic synchronous data transfer over sockets is low in implementation cost. Comparatively, supporting the wide range of fast and modern transfer methods available poses a significant implementation overhead. Ideally, these would additionally be combined with asynchronous data transfer and congestion control, depending on serialization speeds on both ends of the data migration process.

Transfer Formats: The transfer format in data migration is the format in which data gets sent over the wire and to and from which source and destination formats have to be serialized. In a direct one to one connection, the transfer format may

be one of the native formats of the two database systems participating, otherwise there may be additional intermediate formats.

The format influences serialization costs from and into the format itself, depending on the similarity of the other formats. For example, two row encoded database formats may be serialized by rearranging data in place, as both formats already are structured similarly. On the contrary, serializing between a row and a column encoded format encompasses restructuring the data memory layout, which is more cost-intensive.

Different formats have different compression characteristics, which can influence the total bytes needed to be transfered. In general, column encoded formats are more compressible, as same data types are grouped and can be compressed together. String encoded formats, like CSV, are bloated compared to their binary equivalents.

We have to consider these aspects for choosing or recommending different transfer formats for data migration. We need that third transfer format to fulfill our generalization requirement, which aims to decouple participating database systems from each other, and therefore, decouple the serialization processes by decoupling the database formats with a third transfer format.

Implementation Cost: The cost of implementing a transfer method lies in implementing the serialization between different formats, handling data requests and data transfer, supporting additional features like compression and conforming to the database internals. The main attraction of generic data transfer solutions is reducing the implementation cost to a minimum. CSV is widely supported and therefore needs only small setup scripts to export, transfer and import tables into another database. JDBCs common interface allows utilization of already existing drivers, only needing the implementation of a database extension to access the JDBC interface and deserialize the ResultSet format. Generic solutions in general utilize portable connection methods and data formats to replace custom implementations. They expose interfaces, which are implementation friendly and handle much of the implementation work before hand, at the costs of customization and optimization. On the other hand, it is difficult to define an upper limit to the implementation costs for specialized transfer solutions.

In case of PostgreSQL and Clickhouse one can see the how the implementation effort for specialized data wrappers can grow based on given database system features. If one was to implement access from PostgreSQL to Clickhouse, one could use the extension feature of PostgreSQL to implement the data wrapper, which would have access to the internals of the PostgreSQL database system. The wrapper could also access Clickhouse's exposed native interface and therefore had the potential to be an optimal data wrapper with only one serializa-

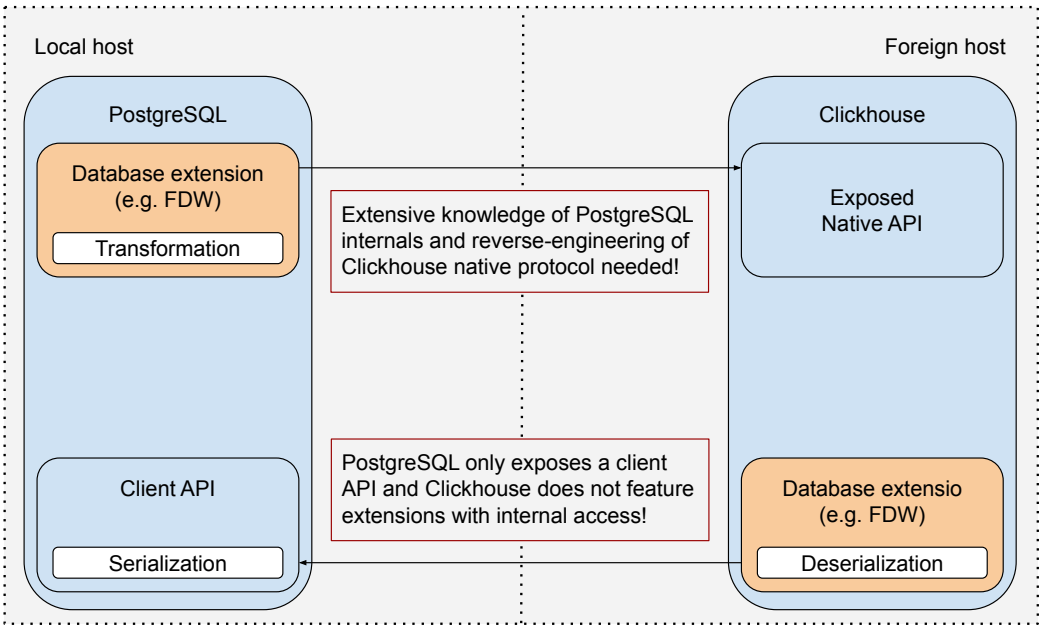


Figure 5: Problems of specialized data wrappers between PostgreSQL and Clickhouse. The ideal implementation of specialized data wrappers to access Clickhouse from PostgreSQL and vice versa comes with immense implementation overhead and even needs adjustments to the source code of both database systems.

tion step needed (see Figure 5). Ideal conditions for a specialized data wrapper. Still a developer would need profound knowledge of the intricacies of the PostgreSQL internals. Additionally, Clickhouse exposed native format protocol does not have a formal specification yet and needs to be reverse-engineered from its source code [Clickhouse(2022)]. The other way around would need an even greater implementation overhead. A custom built connection from Clickhouse to PostgreSQL would need to solve the two big problems, that Clickhouse does not feature extensions with access to its internals and PostgreSQL does not expose its native data format. The solution would need to modify the source code of both database systems and create custom compiled systems, which is rarely feasible or desired. Clickhouse itself already provides the ability to query foreign PostgreSQL servers. But because PostgreSQL only exposes a client interface, Clickhouse needs to access PostgreSQL's client interface and introduces an additional serialization step and a bloated transfer format.

The trade off in implementation costs lies in the question of which parts can

be generalized before hand and at what performance cost or lack of optimization. Serialization to the local native database format can not be generalized, because the native database format differs for each database system. But by introducing a transfer format, which generalizes the data transfer part of the migration, there is only one serialization needed to the native format, instead of multiple serializations, one for each other foreign database format available. As we already aim for a generic solution for data migration, our concern here lies in what level of generalization we want to enforce.

Requirements: In summary, we need to be careful, not to introduce unnecessary serialization steps in our effort to provide a generic architecture. Find solutions to deal with the implementation cost of fast data transfer. Consider performance and similarity aspects of different data formats. And find compromises between implementation costs and optimizations.

4 Warpdrive: Efficient and Generic inter-DBMS Data Movement

The main concerns for developing the Warpdrive architecture are generality, efficiency and ease of use. Generality means, every database system should be connectible via the Warpdrive architecture. Efficiency means a data transfer performance comparable to specialized data wrappers. Ease of use means, that software engineers only have to handle the intricacies of their own database systems to connect these systems to a data federation powered by Warpdrive. Additionally, we aim for SQL/MED compliance, as we are convinced, that the use of an SQL standard eases the implementation aspect of a generic solution and allows Warpdrive to be an easily installed drop-in solution replacing existing wrappers.

In this chapter, we first describe the high-level architecture of Warpdrive and point out the connection between our architecture design decisions and the main concerns and then continue with sub chapters explaining the intricacies of our Warpdrive prototype.

4.1 An Architecture for Generic and Efficient Data Transfer

The Warpdrive architecture connects an arbitrary number of database systems via two connectors each in a simple client/server setup. The client is usually a foreign data wrapper responsible for querying foreign databases, while the server is a database extension or local application communicating with the database and answering queries it receives. An overview of the basic structure of a query running through the Warpdrive is given by Figure 6. First, the query is given to the foreign data wrapper by the SQL/MED API. Second, the foreign data wrapper starts communicating with the database extension. Third, the extension queries its local database system, receives the query results, serializes it and sends the result back to the foreign data wrapper. Fourth, the data wrapper deserializes the result and provides it to the local database system through the SQL/MED API.

The use of the SQL/MED standard is not mandatory. A database extension accepting queries and forwarding results back would be able to fulfill the duties

of a foreign data wrapper and be compatible with the Warpdrive. Nevertheless, utilizing the SQL/MED standard has several important benefits. By supporting an SQL standard for foreign database access, we hope to ease implementation of the client connector in the future, as more systems become SQL/MED compatible. Queries for foreign data wrappers are also standardized, which allows easy utilization of these database systems in decentralized query execution, as an orchestrator can use the same table syntax on each system for accessing foreign systems.

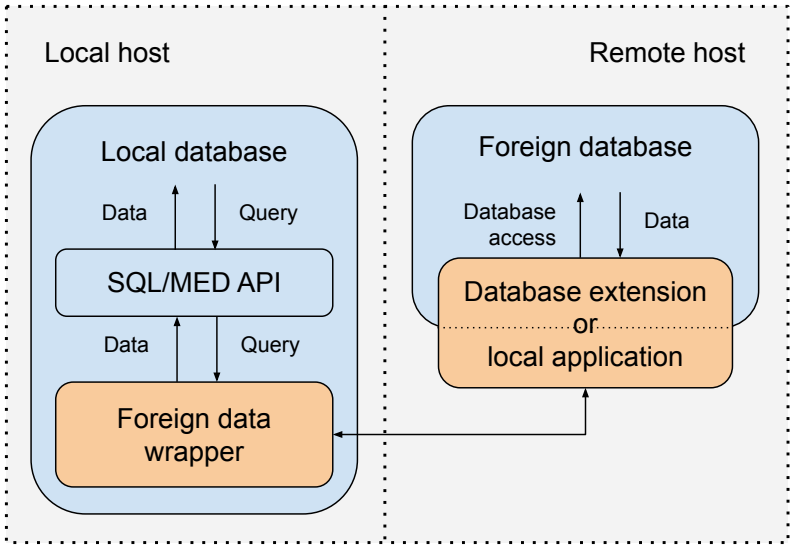


Figure 6: Overview of a query running through Warpdrive. Orange signals developer provided modules and blue signals database systems.

To achieve decoupling for the desired generalization requirement, we pushed the serialization steps onto each participating database system and use a third data format for transferring the data (see Figure 7). Programmers implementing the foreign data wrapper now only have to deal with interactions between their native database format and the transfer format, which is static, in contrast to the different formats used by the different foreign database systems. This highly reduces implementation cost with only one serialization to be implemented in the foreign data wrapper. As serialization steps depend on the underlying database system, but data transfer in a third transfer format does not, we were able to prebuilt the data transfer and separated it in a Warpclient and Warpserver library for usage in specialized Warpdrive data wrappers. This architecture allows heavy data transfer optimization with a specialized format and also heavy serialization

optimization, due to the strong locality of the serialization to the corresponding database system.

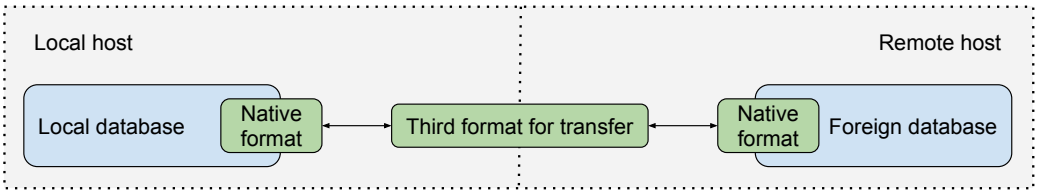


Figure 7: Using a separate format for data transfer essentially decouples database systems from each other. Serialization is only needed from and into the transfer format.

Even though, our architecture assumes the data to be sent in a predefined transfer format, in reality our Warpclient and Warpserver libraries are format agnostic. We recommend to settle for predefined transfer formats in a Warpdrive data federation but only programmed the raw data transfer in our libraries. Systems could settle for a column transfer format like Apache Arrow and require clients and servers to compress the data before using the Warpclient and Warpserver for transmission, as to minimize the raw bytes send over the wire. Or they settle for two formats, one column and one row oriented, as we recommend, and request the appropriate format each time, depending on the interacting database systems. In essence, a format agnostic data transfer gives room for custom performance optimizations, aligned to the needs of different data federations.

We strove for a highly optimized data transfer in our pre-built libraries, because the implementation overhead of the libraries is independent from that of the data wrappers. Currently, we support different remote direct memory access methods, InfiniBand, basic TCP sockets and different shared memory methods, but only in synchronous mode.

The architecture now consists of three parts. The different participating database systems, a de- and serialization extension local to the corresponding database system utilizing the Warpclient and Warpserver library for requesting data from foreign systems (see Figure 8).

This architecture resembles the type of specialized data wrappers, in which the foreign database system only exposes client interfaces and therefore need two serialization steps. These serialization steps of specialized data wrappers can be as optimized as in the Warpdrive architecture, but the exposed client format may be inefficient as transfer format and software engineers have to deal with the data transfer and specific data format of foreign database systems, because the deseri-

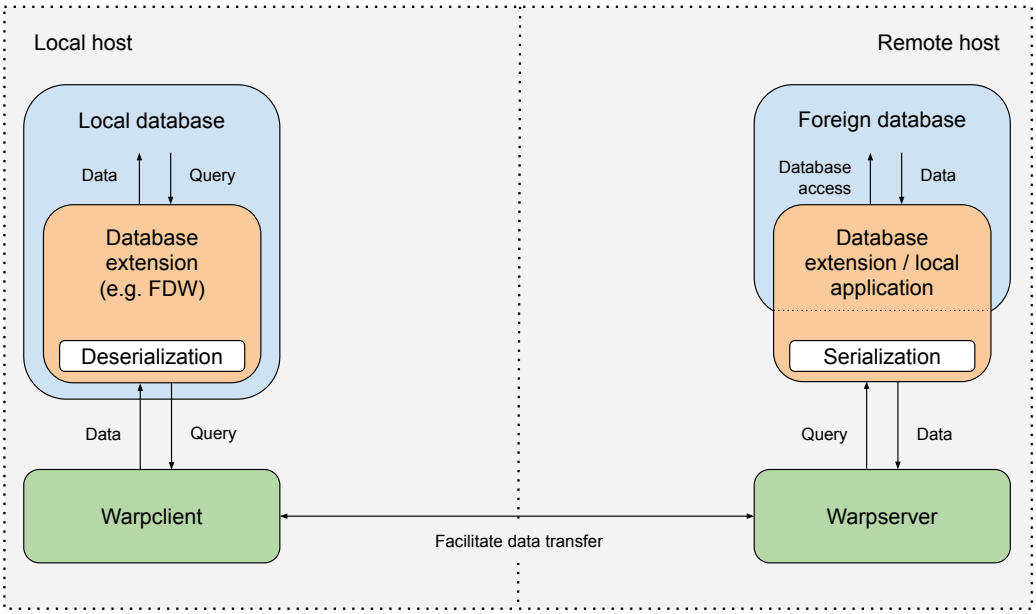


Figure 8: Architecture of foreign data access via Warpdrive. Green signals Warpdrive provided modules, orange signals developer provided modules and blue signals database systems.

alization is not decoupled from the foreign database system. In theory, Warpdrive already has the potential to outperform these kinds of specialized data wrappers. The JDBC architecture, on the contrary, also decouples the software engineer from foreign database systems, but the serialization has to convert to and from the unfit JDBC ResultSet format provided by a transfer system optimized for transactional load.

4.2 Warpdrive Components

Software engineers need to implement two parts for their corresponding database system, to allow participation in a Warpdrive network. First, they need to program a database extension, which includes and calls the Warpclient library, usually an SQL/MED foreign data wrapper. We call this extension client. The job of this client is to forward queries directed at it to the Warpclient, to deserialize the returned transfer format and to insert that deserialized data into the database. Second, they need to program an extension or a standalone program with access to the local database system, whichever is more fitting for the corresponding system,

which includes and starts the Warpserver. We call this extension server. The Warpserver calls this server to query the database in case of a request from a foreign Warpclient and awaits serialized data in transfer format from the server. These two extensions work separately. It would be totally feasible to only implement a client and only participate in requesting data from a Warpdrive data federation. In the following, we will describe the Warpdrive program flow from the viewpoint of the client and the server and describe the interface of the corresponding Warpclient and Warpserver library.

4.2.1 Warpclient

The client extension is usually located in the internals of the database system it serves, which is the ideal location for inserting data into a database system as optimized and fast as possible. Through the SQL/MED API, it receives a query from the system and is expected to return the query results. To get the required data, the client calls the Warpclient interface (see Code Fragment 1).

```
1    DEFAULT_PORT              13337
2    DEFAULT_BUFFER_SIZE       4096
3
4    warpclient_queryServer(server_addr_local, port, transferFormat,
       query);
5
6    warpclient_getData();
7
8    warpclient_cleanup();
9
```

Code Fragment 1: Warpclient Interface

This interface provides three functions for requesting and receiving data from a foreign Warpserver. The first function, *warpclient_queryServer()*, is used to determine the foreign server to be queried, together with the current query and additional information, like the desired transfer format. The Warpclient uses that information to build a connection to the Warpserver, transfers the query and received data. The second function, *warpclient_getData()*, returns a buffer full of query result data. The client deserializes that data, returns it to the database system and requests another buffer, until the return value signals the end of the data stream. The third function, *warpclient_cleanup()*, is used to signal the Warpclient, that all the data got processed and it can release any resources left from the interaction. An overview of the interaction between the client and the Warpclient is given by Figure 9. Because SQL/MED only provides syntax standards but no

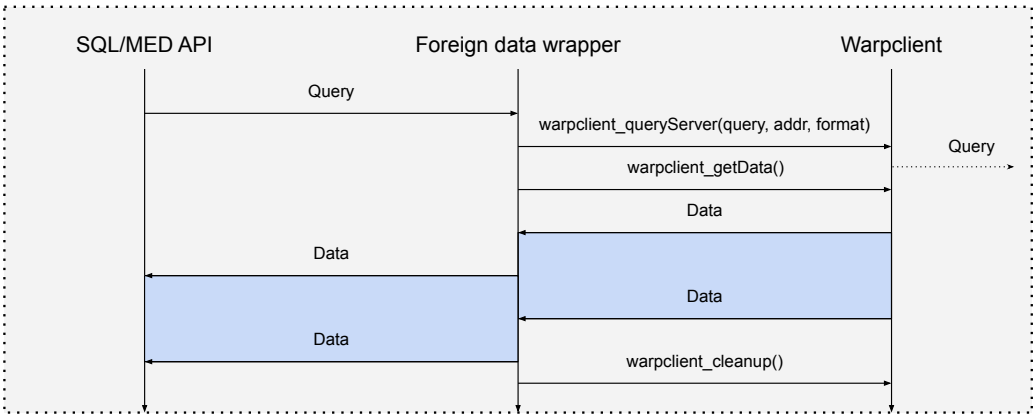


Figure 9: Data flow between foreign data wrapper and Warpclient library.

implementation requirements to database systems, the concrete implementation aspects of the client are dependent on the system it is build for, which makes the client a very specialized implementation, which is not portable, but also allows much room for optimizations.

4.2.2 Warpserver

Depending on the database systems features, the server can be implemented as extension inside the database system, or has to be implemented outside as a standalone application. As a database extension, the server has greater potential for serialization optimization and faster access to the raw binary data of the database system. As a standalone application, the server is easier to implement but lacks performance in comparison.

The Warpserver interface provides three functions and two callback function headers (see Code Fragment 2). The first function, *warpserver_initialize()*, configures the Warpserver and expects two callback functions according to the headers for interaction between the Warpserver and server. The other two functions, *warpserver_start()* and *warpserver_stop()*, start the Warpserver in a separate thread, where it listens for incoming queries. The callback functions allow the server to define, how the Warpserver accesses the database system. As soon as the Warpserver receives a query, it calls the warpserver_cb_query function with the query as parameter and expects this function to query the database system. It then calls warpserver_cb_data in a loop and expects to receive data buffers, which it sends back to the calling Warpclient, until the return buffer signals the end of the result data. Because the Warpserver and server are in separate threads, the server

```
1   DEFAULT_PORT              13337
2   DEFAULT_BUFFER_SIZE       4096
3
4   warpserver_cb_query(transferFormat, querystring);
5   warpserver_cb_data(data, length);
6
7   warpserver_initialize(listen_addr_local, port,
      warpserver_cb_query querycb, warpserver_cb_data datacb);
8
9   warpserver_start();
10
11  warpserver_stop();
12
```

Code Fragment 2: Warpserver Interface

can build elaborate asynchronous data handoffs via these callback functions. E.g. it could fill a buffer pool with result data as soon as the query callback signals a new query and return one buffer each time the Warpserver calls the data callback. An overview of the interaction between the server and the Warpserver is given by Figure 10.

4.3 Warpdrive Prototype

The Warpdrive prototype consists of both libraries, the Warpclient and the Warpserver. They handle the raw data transfer for a given query in a client/server architecture (see Figure 11). Every Warpserver listens for incoming calls from Warpclients. After initial connection buildup, a query message is send from Warpclient to Warpserver, which includes protocol bits, the query length and the query itself (see Table 1). The Warpserver responds with a stream of result data until it signals the end of the data stream and closes the connection.

0	1 … 15	16 … 31
T	Custom protocol bits	Query length
SQL query string		

Table 1: Query message from Warpclient to Warpserver. T: Transfer format bit. 0 for generic column and 1 for generic row format.

To achieve the desired fast data transfer and allow modern data transfer methods, we utilize Unified Communication X (short UCX[Shamis et al.(2015)]). UCX is a communication framework for high-performance network transfer. It allows

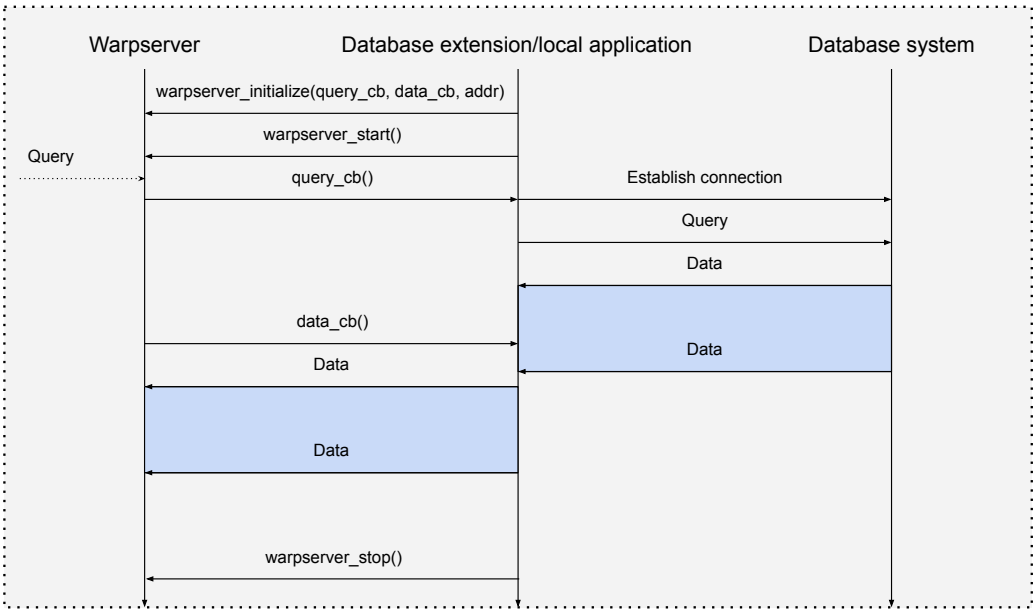


Figure 10: Data flow between Warpserver library and database extension.

abstraction of underlying network hardware capabilities and choses transfer methods depending on available network hardware.

4.3.1 Implementation intricacies

The Warpdrive architecture allows a software engineer to offload the data transfer to the Warpdrive, while only being responsible for the serialization of his database systems format into the transfer format. Still, the programming effort for a client and server is much more comparable to implementing a specialized data wrapper or a type 4 JDBC driver, than more generic type 2 or 1 JDBC drivers. Initially, we planned to micro benchmark the implementation overhead, but it turned out unexpectedly high, that we had to reduce our prototyping to a bare minimum and scrapped the benchmarks.

The offloading of the data transfer responsibility to the Warpdrive lightens the programming effort, but intricate knowledge of the local database system is still essential for a competitive serialization process. This serialization, to be competitive, has to convert between binary data formats, which are seldom well documented. This means, that binary formats need to be reverse-engineered or read from source code. Additionally, the deserialized native binary format has to be

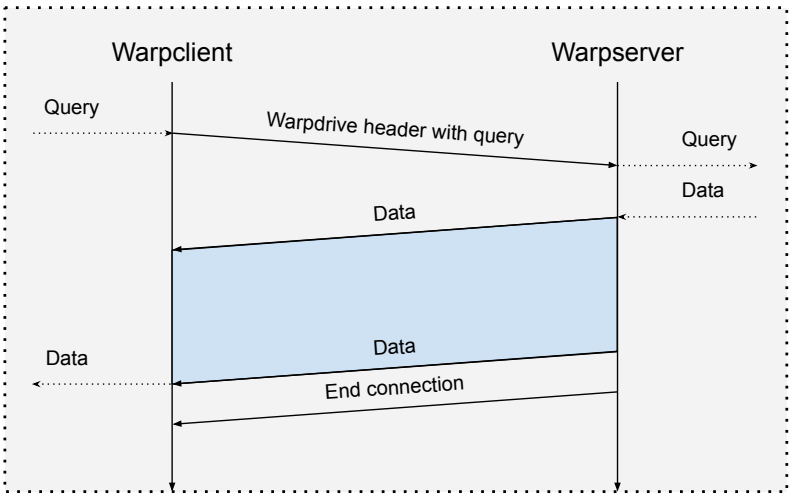


Figure 11: Communication between Warpclient and Warpserver.

delivered into the database in a way the database can handle the data. We are still confident, that our architecture can be an improvement in the generality of efficient data transfer, as the main programming effort only has to be done once for every database system type instead of every connection between system types.

0 … 15	16 … 31
ndigits	weight
sign	dscale
digits	

Table 2: The binary output of a Decimal of PostgreSQL's client format. Every field consist of 2 byte integers. ndigits is the length of the digits field. weight states the number of digits before the decimal point, before the digits got turned into a String representation. dscale states the number of characters behind the decimal point, after the digits got turned into a String representation.

On our work of decoding the PostgreSQL binary Decimal type included in the Lineitem table, the high implementation effort needed for a data wrapper can be seen. Our Warpdrive architecture allows heavy wrapper optimization, as each wrapper, client and server, sits right on top of its database system. The fastest link would be to directly read the databases native binary format. As a fully optimized

PostgreSQL data wrapper was out of scope as an experimental prototype, we settled for the PostgreSQL binary client format, which can be directly requested via queries. Decoding this format still was a lot of work. There is no documentation about the binary client format and it has to be deduced from code. After a week of full time work, we were able to understand the binary Decimal and had a function ready to convert the Decimal into a String representation (see Table 2). Feeding the binary Decimal representation back into the database system was more difficult, as we had to find internal functions responsible for handling specifically binary client format decimals and call them in a PostgreSQL specific way, which went quickly out of scope for this thesis. While it is clearly feasible for an optimized client and server wrapper participating in a Warpdrive network to handle binary native formats, and we made progress on that front, it was not feasible for an experimental prototype.

5 Performance Evaluation

We implemented a basic foreign data wrapper client prototype for PostgreSQL and a server prototype for PostgreSQL and Clickhouse for testing our Warpdrive prototype implementation. In our experiments we compare different data transfer mechanisms with our client/server and Warpdrive prototype to evaluate the performance of our solution compared to state-of-the-art data transfer. In the following chapters we describe our experimental setup and analyze the performance test results .

5.1 Experimental Setup

Hardware: All experiments are conducted on a server running 16 Intel(R) Xeon(R) E5530 cores with a clock rate of 2.40GHz and 47GB of system memory.

Software: We are using PostgreSQL and Clickhouse for our experiments. All database systems and the test suit are run in Docker containers with Docker version 20.10.17 and Docker-compose version 1.29.1. The PostgreSQL image is based on the Docker image ubuntu:focal (official Ubuntu 20.04 image) and uses PostgreSQL 12 installed from the Ubuntu repositories. The Clickhouse image is the official Clickhouse image for testing the Clickhouse-JDBC-Bridge, it is based on the image Clickhouse/Clickhouse-server:22.3. Further dependencies of the test project files are all available in the Git repository of the Warpdrive [Ziegler(2022)].

Data: In our experiments we are transferring TPC-H table data from two datasets, SF1 and SF10. These datasets are generated via TPC-Hs dbgen utility. We only transfer columns with data types identifier (64 Bit integer), integer (32 Bit integer), fixed and variable sized text from the Lineitem table. This amounts to nine columns and results in a table size of 515 Megabyte for Lineitem SF1 and 5.2 Gigabyte for Lineitem SF10. Queries are given at the test cases below.

Methodology: We transfer data between pairs of systems. We measure the time for the data transfer by storing the systems nanoseconds via Javas *System.nanoTime()* method before and after we issue the transfer query and take the difference. The correctness of the data transfer gets validated by comparing

the line count of source and target table after initial verification, that the transfer works.

Reproducibility: The Git repository[Ziegler(2022)] contains all code for the Warpclient and Warpserver library prototype, the foreign data wrapper client and server prototype implementations, the Docker files, all used JDBC drivers and foreign data wrappers and the test suit code. Additionally, it contains Makefiles and scripts together with Readme files giving instructions on how to run the experiments. The only software needed is a Docker and docker-compose installation compatible with the version we used and an installation of Make.

5.2 Performance for Row-Based Systems

In our first test case we transfer the reduced Lineitem table between two PostgreSQL instances to compare state-of-the-art transfer methods with our Warpdrive prototypes. We use the SQL query given in Code Fragment 3. The used transfer methods are: CSV file transfer, third party JDBC foreign data wrapper implementation by [Sharma(2022)] and the native PostgreSQL foreign data wrapper. The results are given by figure 12.

```
1    INSERT INTO local_lineitem_sf1
2    SELECT
3    l_orderkey, l_partkey, l_suppkey,
4    l_linenumber, l_returnflag, l_linestatus,
5    l_shipinstruct, l_shipmode, l_comment
6    FROM lineitem;
7
```

Code Fragment 3: Query for transferring Lineitem SF1

We expected to see a major difference between the generic transfer methods, CSV and JDBC, and the native PostgreSQL foreign data wrapper, with our prototype close to the generic solutions. Contrary to our expectations, CSV, native and our prototype are similar in performance, with JDBC performing multiple times worse and even not capable of finishing the SF10 transfer before our ssh connection to the cluster times out, which is around half an hour. From the three methods performing comparably, the native one takes 21.1% respectively 23.7% more time to finish the data transfer than the fastest method, while CSV is fastest at SF1 and the Warpdrive is fastest at SF10.

We assume the bad performance of the PostgreSQL native foreign data wrapper is due to the use of their client protocol libpq and a synchronous sending archi-

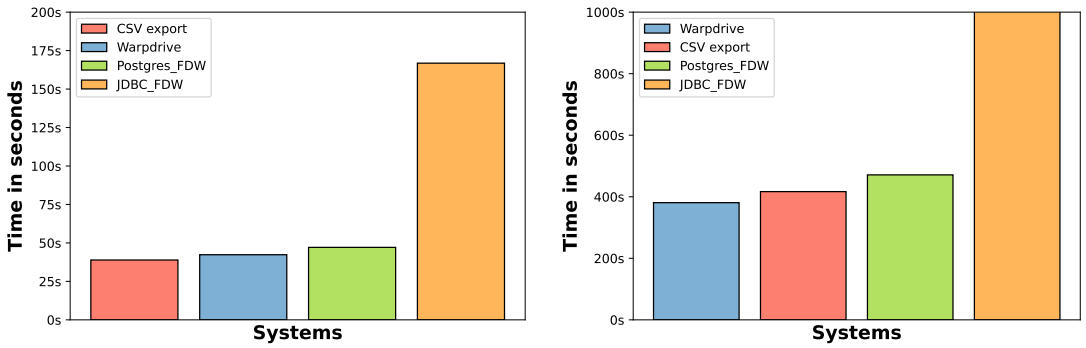


Figure 12: Result of Lineitem transfer between PostgreSQL instances. Left from SF1 and right from SF10 data set.

tecture with plain sockets. This introduces two serialization steps into the data transfer, where none would be needed between two database systems of the same type. The first step, as the foreign server serializes its data into the client format and the second step as the foreign data wrapper deserializes the format back to the native. Additionally, libpq can return two kinds of client formats, one binary and one non-binary. The foreign data wrapper uses the non-binary version, which potentially increases serialization cost and also format bloat, which reduces socket transfer speeds.

Our client and server prototypes are build quite similarly. Our server queries its local PostgreSQL instance via libpq and our Warpdrive prototypes data transfer is synchronous. But contrary to the native foreign data wrapper, we utilize the binary client format. Combined with our data transfer, which, while synchronous, is optimized and reduces memory copies to a minimum, we got an overall faster data transfer with a smaller data size footprint.

The CSV transfer performance ranking is mostly due to suboptimal performance of the foreign data wrapper and our client prototype. While easy to manage, there is manual labor involved in the transfer process, which increases the attractiveness of the other methods over CSV.

5.3 Performance between Row- and Column-Based Systems

In our second test case we transfer data from a Clickhouse instance to a PostgreSQL instance to verify the generic architecture capabilities of our Warpdrive, as we reuse the same client prototype for the PostgreSQL end of the data transfer. This test case differs from the first, in that Clickhouse is a column based database system,

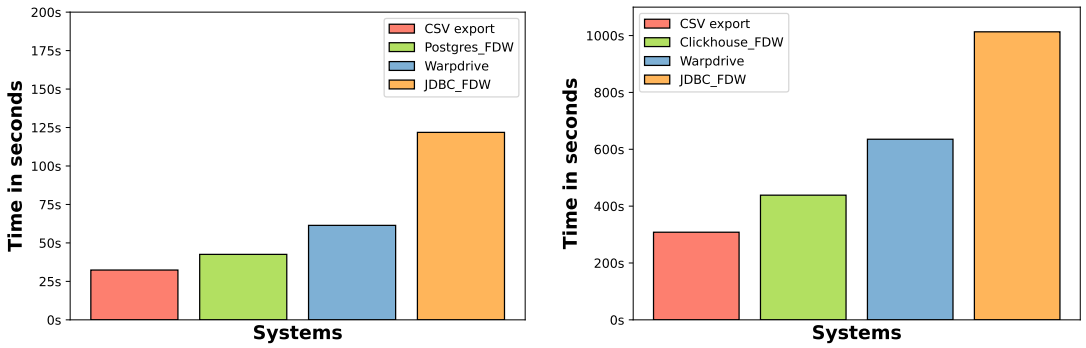


Figure 13: Result of Lineitem transfer between Clickhouse and PostgreSQL. Left from SF1 and right from SF10 data set.

whereas PostgreSQL is a row based system. Serialization between row and column data formats is costlier, as the formats differ more than two row formats. Again, we use a CSV data transfer and a JDBC foreign data wrapper as state-of-the-art generic data transfer and a Clickhouse foreign data wrapper as state-of-the-art specialized data transfer. We use a query similar to test case one, only changing server addresses. The results are given by figure 13.

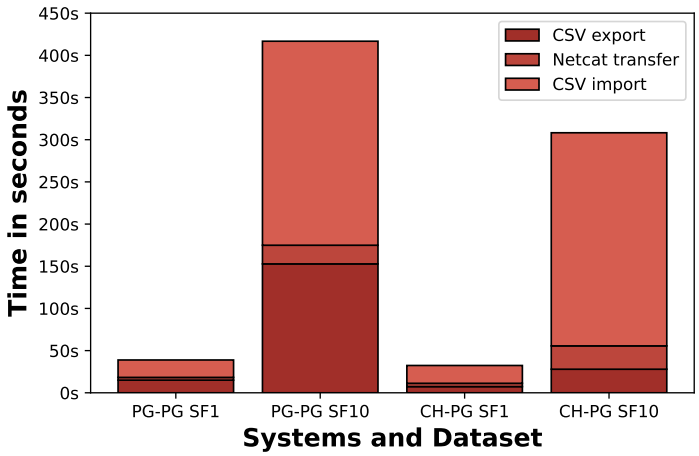


Figure 14: Clickhouse CSV file export performance scales exceptionally good compared to PostgreSQL

The results are mostly in line with our expectations, as JDBC and the CSV import are slow in comparison to the Clickhouse foreign data wrapper. Our Warpdrive prototype performs in between the two types of data transfer. Surprisingly, the CSV file export of the Clickhouse server was extremely fast, to the point of

plain copying, even scaling better than the Netcat file transfer (see figure 14). We assume, that Clickhouse optimized the CSV export in a way, that there is minimal serialization needed. This export performance places CSV at the top. Overall, we are surprised, that CSV file transfer is still a competitive data transfer method, given its reliability, which was not given with generic JDBC transfer.

The difference between the Clickhouse foreign data wrapper and our Warpdrive prototype in performance can be attributed to serialization differences. As Clickhouse is optimized to export data in numerous formats, one of which is PostgreSQLs client format, the foreign data wrapper likely accesses this format directly. Whereas our prototype queries the Clickhouse native format and serializes it into the PostgreSQL client format, which we choose, because in a larger data federation the native Clickhouse format would be the starting point to serialize into a common transfer format.

6 Related Work

Multiple research work exist, dealing with the problem of data migration. As specialized solutions are usually not portable between systems, research work on this topic focuses on generic solutions with different gradations. We will take a look at generic architectures Muses and Portage, data transfer pipe generator Pipegen and BigDAWG improvement FastDAWG, put them in relation to our work and discuss the differences.

Muses: Muses [Kaitoua et al.(2019)] is an architecture for distributed data migration. It decouples database systems from each other by the use of Connectors and Muses Nodes, which all get their tasks distributed by a Muses Manager. To migrate data between two systems, the Muses Manager first delivers instructions to participating Muses Nodes, which control their Connectors. The Connector reads data from the local database system and serializes it into a Muses stream format, which encapsulates an Apache Arrow stream. This stream gets sent over to other Muses Nodes and their Connectors, which deserialize the stream and insert the data into their corresponding database system. Each Muses Node can have multiple Connectors for their local database systems.

While the data migration use case of Muses is similar to our use case, we differ in important aspects. Muses enforces the Apache Arrow columnar data format as transfer format. While Apache Arrow is optimized for network transfer and in memory serialization, which is ideal for data migration, there are scenarios in which a columnar transfer format is not optimal. Migrating data between two row-based systems would incur high serialization costs twice, which could be reduced by using a row-based transfer format.

Portage: Portage [Dziedzic et al.(2016)] is a data migration framework outlining and separating distinct steps in the extract-transform-load process. It divides the data migration into four components, the Extract, the Transformers, the Migrator and the Import component, each has to be implemented from a software engineer. It provides an overview of the influence of transfer formats to serialization and transfer performance, but does not address the implementation overhead associated with direct binary serialization compared to intermediate transfer format serialization. Both these kinds of data transfer are realizable with Portage, as the

framework imposes little restraints to software engineers, but the concrete Migrator architecture is not provided. In contrast to Portage, we impose more restraints onto the software engineer, as we are a generic data migration architecture, but we also provide architectural solutions for data migration.

Pipegen: Pipegen [Haynes et al.(2016)] modifies Java written database systems to allow generic data migration in a predetermined transfer format. It exploits on the common CSV format, which most database systems have an export and import function for, analyzes the Bytecode for these functions and rewrites these functions with data migration pipes. In their experiments, they compare different transfer formats, but ultimately are bound to one format. Pipegen cleverly circumvents the high implementation overhead for serialization. Their Java and unit test requirement make Pipegen come short as a generic data migration solution.

FastDAWG: FastDAWG [Yu et al.(2019)] is an improvement to BigDAWGs data migration [Gadepally et al.(2016)]. BigDAWG builds on Portage to connect its database systems, but provides relatively little detail about the actual implementation aspects. The architecture describes islands of database systems. Each system has a wrapper between its data format and the island format. Additionally, there are wrappers between islands. Essentially, BigDAWG is a mix between specialized wrappers between a small number of islands and generic wrappers for higher numbers of database systems in an island. While the generic island connections benefit greatly from the grouping of similar data format database systems, as serialization cost is decreased, sending data across islands requires three serialization steps.

FastDAWG identifies the implementation overhead for writing the wrappers needed to incorporate additional database systems as too costly and criticizes slow data migration performance overall. The solution is reducing full query potential only to a main system, and allowing only a subset of queries to be performed on additional systems. This still allows full access to data stored in all systems, but essentially converts the island into a Mediator/Wrapper architecture.

Our Warpdrive aims to ease the development of generic wrapper solutions and could elevate BigDAWG, erasing the need for compromise solutions like FastDAWG.

7 Conclusion and Future Work

We worked on the problem of data migration, which is a major factor in many data analytics use cases and also important for rising architectures seeking to utilize database specific computation advantages. Existing generic solutions fail to meet performance requirements and specialized solutions are difficult to set up, not reliable to find and must be specifically crafted for larger polystore systems.

First, we analyzed existing work and identified the main aspects of data migration, namely serialization steps and the implementation overhead that comes with it, data transfer and the transfer format influencing both. From that knowledge, we designed an architecture called the Warpdrive with the aim to deliver a generic data migration framework for arbitrarily large data federations and being SQL/MED compatible for decentralized query execution. We made sure, that serialization is kept in check to allow good performance potential, by pushing serialization steps near the two participating databases of a transfer and capping the number of serialization steps, while allowing generality by staying format agnostic and decoupling database systems from each other. Finally, we implemented Warpdrive library prototypes, which ease the task of adding wrappers for the Warpdrive. To test our architecture and implementation, we implemented prototype wrappers for PostgreSQL and Clickhouse, which already outperformed JDBC and went on par with PostgreSQL's native foreign data wrapper.

Our Warpdrive architecture has the potential to be used as generic transfer mechanism and as connection mechanism to allow free data transfer between database systems. Its usefulness depends on future wrapper implementations and inclusion in future scientific work. As our prototype wrappers and also the prebuilt Warpclient and Warpserver libraries are heavily underdeveloped, there is much potential for performance gains. Future work should start with enhancing the Warpdrive libraries data transfer, as it still lacks asynchronous, parallel transfer and UCX and PostgreSQL memory buffers were not used properly for performance gains. On the scientific end, there is room for further experiments with different transfer formats, which we initially planned in this thesis. The client and server prototypes could be extended with more data types, serialize into a transfer format and properly utilize the locality to their database systems to read/write data with internal database functions instead of client interfaces. These additions should increase the performance of the prototypes by a great margin. Additionally, we could get more insight into optimal transfer formats between row and column

based systems by experimenting with the optimal point of serialization from row to column format in the data transfer.

Bibliography

[Andersen(2017)] Lance Andersen. 2017. JDBC™ 4.3 Specification. Retrieved 2022-08-25 from https://download.oracle.com/otn-pub/jcp/jdbc-4_3-mrel3-eval-spec/jdbc4.3-fr-spec.pdf?AuthParam=1661433131_97389f5580520dc7b0142fd13a64d321

[Bondiombouy and Valduriez(2016)] Carlyna Bondiombouy and Patrick Valduriez. 2016. Query processing in multistore systems: an overview. (2016), 32.

[Clickhouse(2022)] inc. Clickhouse. 2022. Native Interface (TCP) | ClickHouse Docs. Retrieved 2022-09-14 from https://clickhouse.com/docs/en/interfaces/tcp

[Dziedzic et al.(2016)] Adam Dziedzic, Aaron J. Elmore, and Michael Stonebraker. 2016. Data transformation and migration in polystores. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Waltham, MA, USA, 1–6. https://doi.org/10.1109/HPEC.2016.7761594

[Gadepally et al.(2016)] Vijay Gadepally, Peinan Chen, Jennie Duggan, Aaron Elmore, Brandon Haynes, Jeremy Kepner, Samuel Madden, Tim Mattson, and Michael Stonebraker. 2016. The BigDAWG polystore system and architecture. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–6. https://doi.org/10.1109/HPEC.2016.7761636

[Haynes et al.(2016)] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2016. PipeGen: Data Pipe Generator for Hybrid Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, Santa Clara CA USA, 470–483. https://doi.org/10.1145/2987550.2987567

[Kaitoua et al.(2019)] A. Kaitoua, T. Rabl, A. Katsifodimos, and V. Markl. 2019. Muses: Distributed Data Migration System for Polystores. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1602–1605. https://doi.org/10.1109/ICDE.2019.00152 ISSN: 2375-026X.

[Manyika et al.(2011)] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, Angela Hung Byers, et al. 2011. *Big data: The next frontier for innovation, competition, and productivity.* McKinsey Global Institute.

[Melton et al.(2002)] Jim Melton, Jan Eike Michels, Vanja Josifovski, Krishna Kulkarni, and Peter Schwarz. 2002. SQL/MED: a status report. *ACM SIGMOD Record* 31, 3 (Sept. 2002), 81–89. https://doi.org/10.1145/601858.601877

[Mohammadpoor and Torabi(2020)] Mehdi Mohammadpoor and Farshid Torabi. 2020. Big Data analytics in oil and gas industry: An emerging trend. *Petroleum* 6, 4 (Dec. 2020), 321–328. https://doi.org/10.1016/j.petlm.2018.11.001

[Raasveldt and Mühleisen(2017)] Mark Raasveldt and Hannes Mühleisen. 2017. Don't hold my data hostage: a case for client protocol redesign. *Proceedings of the VLDB Endowment* 10, 10 (June 2017), 1022–1033. https://doi.org/10.14778/3115404.3115408

[Shamis et al.(2015)] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 40–43. https://doi.org/10.1109/HOTI.2015.13 ISSN: 2332-5569.

[Sharma(2022)] Atri Sharma. 2022. JDBC_FDW. Retrieved 2022-10-04 from https://github.com/atris/JDBC_FDW Retrieved August 2, 2022.

[Yu et al.(2019)] Xiangyao Yu, Vijay Gadepally, Stan Zdonik, Tim Kraska, and Michael Stonebraker. 2019. FastDAWG: Improving Data Migration in the BigDAWG Polystore System. In *Heterogeneous Data Management, Polystores, and Analytics for Healthcare*, Vijay Gadepally, Timothy Mattson, Michael Stonebraker, Fusheng Wang, Gang Luo, and George Teodoro (Eds.). Vol. 11470. Springer International Publishing, Cham, 3–15. https://doi.org/10.1007/978-3-030-14177-6_1

[Ziegler(2022)] Joel Ziegler. 2022. Warpdrive. Retrieved 2022-10-04 from https://github.com/Hyrikan/Warpdrive