Enter the Warp: Fast and Adaptive Data Transfer with XDBC

Haralampos Gavriilidis
TU Berlin, BIFOLD
Germany
gavriilidis@tu-berlin.de

Joel Ziegler
TU Berlin
Germany
joel.ziegler@campus.tu-berlin.de

Midhun Kaippillil Venugopalan
TU Berlin
Germany
m.venugopalan@tu-berlin.de

Benedikt Didrich
TU Berlin
Germany
b.didrich@campus.tu-berlin.de

Matthias Boehm
TU Berlin, BIFOLD
Germany
matthias.boehm@tu-berlin.de

Volker Markl
TU Berlin, BIFOLD, DFKI
Germany
volker.markl@tu-berlin.de

ABSTRACT

Fast and scalable data transfer is crucial in today's decentralized data ecosystems and data-driven applications, including extraction-transformation-loading (ETL) pipelines, and data science workflows. Transfers often occur across heterogeneous environments—ranging from cloud-hosted systems to local consumer devices—with varying compute and network constraints. However, existing solutions struggle to balance performance with generality across such diverse setups. We recently proposed XDBC, a holistic data transfer framework that decomposes the pipeline into logical components with multiple physical implementations per component. Its modular architecture enables seamless system integration and automatic tuning based on workload and environment characteristics. In this demonstration, we present Enter the Warp, an interactive game built around XDBC that visualizes data transfer as a space mission. Players configure transfer parameters, monitor live throughput metrics, and optimize performance to shield Earth from meteor strikes, gaining an understanding and hands-on experience of data transfer challenges in an engaging and intuitive way.

PVLDB Reference Format:
Haralampos Gavriilidis, Joel Ziegler, Midhun Kaippillil Venugopalan, Benedikt Didrich, Matthias Boehm, and Volker Markl. Enter the Warp: Fast and Adaptive Data Transfer with XDBC. PVLDB, 18(12): 5315 - 5318, 2025.
doi:10.14778/3750601.3750660

1 INTRODUCTION

Today's data-driven applications rely on multiple data systems for storage and processing. Use cases such as machine learning (ML) [12] and extraction-transformation-loading (ETL) [10, 11] pipelines, as well as cross-platform and federated queries [5, 9], require transferring data across system boundaries. These systems operate in diverse environments, including cloud platforms, on-premise infrastructure, and consumer devices. As data ecosystems grow, efficient yet generic data transfer across heterogeneous environments and systems becomes increasingly important.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750660

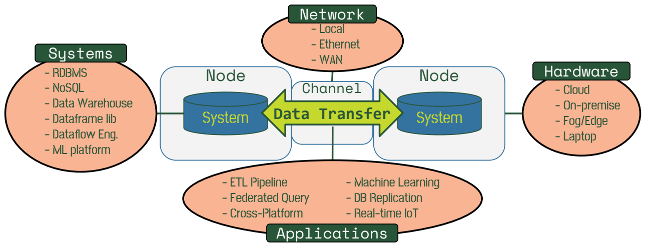


Figure 1: Data Transfer Across Heterogeneous Topologies.

Challenges of Cross-System Data Transfer: Efficient data transfer is difficult due to the heterogeneity of systems, data formats, and deployment environments. Solutions typically rely on specialized connectors, which offer high performance but limited flexibility; or generic drivers like JDBC/ODBC, which are broadly compatible but inefficient. Data transfer involves multiple stages—reading, (de)serialization, (de)compression, and transmission—that must be tuned to the data and environment characteristics. Striking the right balance of generality and specialization, while adapting to environment characteristics, remains a key challenge.

Limitations of Existing Work: Prior work addressed ETL optimization [10, 11] and integration flows [3], but efficient cross-system data transfer is less explored. Recent work showed that JDBC-like approaches are inefficient due to the row-based design and metadata overhead [8]. Approaches like Pipegen [7], Zigzag joins [12], and ConnectorX [13] optimize specific use case scenarios but lack generality. A broadly applicable framework that adapts to heterogeneous systems and environments was missing.

XDBC: A Holistic Data Transfer Framework: To address these challenges, we recently introduced XDBC [4, 6], a modular framework for efficient and adaptive data movement. XDBC provides general reader and writer interfaces for integrating arbitrary data systems and decomposes transfers into configurable components (e.g., read, serialize, compress, transmit). Our heuristic optimizer then automatically selects effective configurations based on workload and environment characteristics. We found that decomposing the pipeline allows for better adaptation to environments, and XDBC outperforms generic data transfer tools by up to 5×, while matching the performance of specialized connectors.

Demonstration: Data Transfer as an Interactive Game: To study the impact of data transfer optimization, we present an interactive *game-based* demonstration. Users take on a mission to ensure efficient data movement across different environments, where

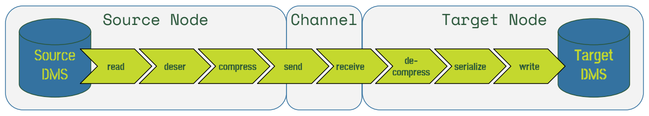


Figure 2: Data Transfer Anatomy.

transfer speed determines the strength of a protective shield around Earth. Each level simulates a unique data transfer scenario with different systems, hardware capabilities, and network speed. Before the transfer begins, users configure key parameters and observe their effects in real time as data flows between a base station and a spaceship. By interacting with live metrics and adjusting settings dynamically, users develop an intuitive understanding of how different factors impact data transfer efficiency.

2 MOTIVATION AND BACKGROUND

Next, we present common data transfer applications, discuss data transfer internals, and outline the limitations of current approaches.

2.1 Data Transfer Applications

Data transfer naturally occurs between diverse systems, e.g., RDBMSes, NoSQL stores, data warehouses, dataflow engines, and ML platforms, as shown in Figure 1. These systems operate in variety of hardware environments (cloud, on-premise, edge) and communicate over different networks (local, WAN, high-latency links).

Example Applications: A common data science use case is loading data from PostgreSQL into pandas (e.g. for ML pre-processing). Users may employ a *specialized* solution like ConnectorX [13], which optimizes parallel data extraction; or a *generic* approach such as turbodbc [2], which internally relies on ODBC. Even within this single use case, environments vary: pandas may run locally on a laptop with limited compute resources and a slow network connection to a cloud-hosted DBMS, or it may run in a cloud-hosted Jupyter notebook (e.g., Google Colab) with scalable compute power and high-bandwidth connectivity. Similar transfer scenarios arise in other use cases such as Apache Spark loading data from an RDBMS or migrating data between two PostgreSQL instances. Each scenario imposes different constraints on compute, memory, and network, requiring an adaptable data transfer strategy.

2.2 Anatomy of Data Transfers

Data Transfer as a Streaming Pipeline: Data transfer involves moving data between systems through a sequence of transformations. As shown in Figure 2, this process follows a streaming pipeline with distinct stages. First, data are *extracted* from the source system, e.g., an RDBMS or file storage. Second, data are *deserialized* from the source format into an intermediate format (e.g., row- or column-based). Third, the intermediate data are optionally *compressed* (e.g., through snappy) to reduce transmission overhead. Fourth, the data are *transmitted* over a network, memory channel, or file interface, depending on the deployment scenario. On the receiving side, the data is *decompressed* (if necessary) and *serialized* back from the intermediate format into the target format. Finally, data are *written* into the target system, e.g., a DataFrame or RDBMS.

Tuning for Different Environments: While this pipeline remains conceptually similar across applications, the ideal configuration depends on system characteristics. For example, compression may improve performance in bandwidth-constrained environments (e.g., a laptop connected to a cloud DBMS) but introduce unnecessary overhead in high-throughput cloud-to-cloud transfers. Similarly, the intermediate layout (i.e., row- vs. column-based) and buffer size affect performance depending on the workload and system architecture. For effective data transfer, one should holistically tune these components to match the underlying execution environment.

2.3 Existing Approaches and their Limitations

Specialized Data Transfer Solutions: Specialized connectors are designed for specific system pairs, optimizing performance through direct end-to-end integration. For example, Connector-X [13] efficiently loads database tables into pandas DataFrames by parallelizing queries and reducing serialization overhead. Similarly, PostgreSQL Foreign Data Wrappers (FDWs) [1] allow cross-DBMS communication by exposing external databases as virtual tables. While these approaches are efficient, they require substantial engineering effort per system pair and lack adaptability when deployed in different network or hardware environments.

Generic Data Transfer Solutions: Generic solutions such as JDBC and ODBC-based connectors on the other hand provide broad compatibility, allowing various applications to connect to databases without system-specific implementations. For example, Spark JDBC enables Spark to read from transactional databases, while turbodbc optimizes DBMS-to-DataFrame transfers using efficient batch loading [2]. However, these solutions typically rely on row-based transfer formats, incurring metadata overhead [8], and offer limited tuning options. These generic solutions are generally not designed to dynamically adapt to different execution environments, leading to suboptimal performance in different environments.

The Missing Piece: Adaptive Data Transfer: Both specialized and generic approaches do not adjust configurations based on workload and environment characteristics. For example, a PostgreSQL-to-pandas transfer may require different strategies on a local laptop with limited bandwidth versus a cloud-hosted Jupyter notebook with high-speed access. Similarly, transferring data between DBMS instances or from a DBMS to Spark may require tuning intermediate format, compression library, and the degree of parallelism of individual components (read, serialize, compress, write). Existing solutions lack dynamic optimization, leading to inefficiencies in heterogeneous environments. An effective framework should balance *generality* (allowing seamless system integration) with *specialization* (optimizing performance per environment), and should offer a modular architecture that decouples transfer components and supports their independent configuration.

3 XDBC: SCALABLE DATA TRANSFER

XDBC enables fast and scalable data transfer across heterogeneous systems. We now describe its design principles and architecture.

Design Philosophy: XDBC addresses the tradeoff between generality and specialization through a modular and extensible architecture. As shown in Figure 3, the framework follows a client-server model, where data flows from a source system (e.g., a DBMS)

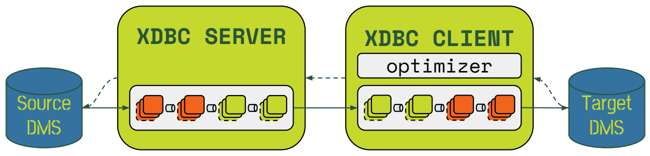
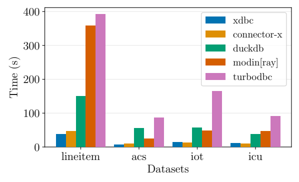
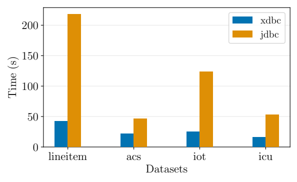


Figure 3: XDBC Architecture Overview



(a) Pandas from PostgreSQL



(b) Spark from PostgreSQL

Figure 4: XDBC Performance Evaluation.

through the *XDBC Server* to the *XDBC Client*, which loads them into the target system. The data transfer pipeline consists of independently configurable components for reading, (de)serialization, (de)compression, transmission, and writing, each supporting multiple physical implementations. XDBC's optimizer selects transfer configurations based on system and environment characteristics, tuning parameters such as compression (e.g., `snappy`, `zstd`), intermediate format (row, column, Arrow), and the parallelism of individual components. This design allows XDBC to provide both plug-and-play extensibility and workload-specific optimizations.

Runtime Architecture: XDBC achieves high performance through efficient memory management and parallelism. We preallocate a fixed memory segment organized as a ring buffer, and pass buffer IDs between components instead of copying data. Each component may utilize multiple workers for improving throughput, while different components operate in a pipeline-parallel manner. Our recent extension supports runtime reconfiguration, allowing components to be dynamically adjusted to changing environments or optimized on the fly. This capability is valuable not only for demonstration purposes but also for practical reconfiguration in real-world scenarios. XDBC implements back-pressure through bounded queues between components, enabling scalable, low-overhead streaming data transfer while gracefully handling contention.

Adaptive Optimization: Since data transfer performance depends on system, dataset, and environment characteristics, XDBC includes a lightweight optimizer that automatically tunes configurations to maximize throughput. Our initial optimizer used a heuristic approach, which worked well in our experiments. We modeled the pipeline as a sequence of components, each with configurable parallelism, and estimated per-component throughput using a cost model derived from offline profiling. Parallelism was incrementally assigned to the slowest component until resource constraints were met. Building on this, we have developed a data-driven optimizer that not only finds better configurations but can also dynamically adapt to changing environments on the fly [4]. This approach enables XDBC to achieve high performance without manual tuning.

Built-in Components and Extensibility: XDBC natively supports common systems and formats such as PostgreSQL, Clickhouse, CSV, Parquet, Spark, and pandas. We utilize custom binary row/column formats and Apache Arrow as intermediate layout, support multiple compression libraries (e.g., `zstd`, `snappy`, `lz4`, `lzo`), and use TCP transfers via `boost.asio`. Furthermore, XDBC is highly extensible: new systems can be integrated by implementing simple read/write interfaces, and new formats or compression methods can be added via pluggable operators, supporting diverse workloads and environments with little effort.

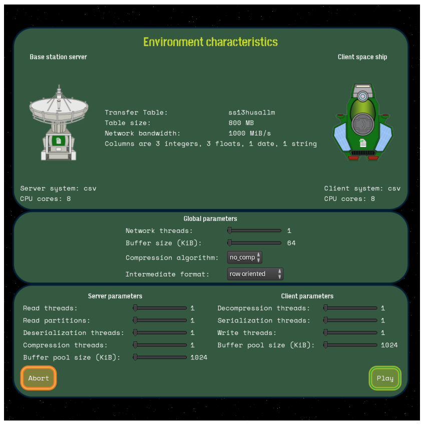
Experimental Findings: In our original work, we have conducted extensive experiments across diverse environments [6]. Here, we highlight two key end-to-end benchmarks. For a data science application (PostgreSQL-to-pandas, Figure 4a), XDBC outperforms generic solutions like `turbodbc` by up to 8× and matches specialized ones like ConnectorX. For an ETL application (PostgreSQL-to-Spark, Figure 4b), XDBC outperforms Spark's generic JDBC data source by up to 5×. Our micro-benchmarks show that individually tuning components is crucial for performance. XDBC's optimizer effectively selects configurations with minimal overhead, demonstrating that adaptive tuning is practical and beneficial. Overall, XDBC bridges the gap between generality and specialization, outperforming generic approaches and matching specialized solutions.

4 DEMONSTRATION: ENTER THE WARP

To make data transfer concepts clear and engaging, we present "Enter the Warp", a custom interactive game built on top of the XDBC framework. In this simulation, Earth is under constant meteor threat, and users must complete data transfer missions to generate a protective shield. Each mission mirrors a real data transfer, requiring users to configure source and target systems, tune parameters, and adapt to environment constraints. Higher throughput translates into faster shielding and better scores. Through gamification, our demo illustrates data transfer concepts, i.e., heterogeneity of environments, pipeline configuration, and runtime behavior, by turning them into interactive and visually intuitive challenges. We now describe the game structure, scenarios, and user interactions.

Data Transfer Environments as Campaigns and Levels: The game is organized into *campaigns*, each consisting of multiple *levels* that reflect different data transfer environments. Each level simulates a unique system setup, varying the source and target systems (e.g., PostgreSQL, pandas, Spark, CSV), as well as the compute capacity of the server (base station, source system) and client (spaceship, target system). The network connecting them is represented as an interstellar link, whose bandwidth fluctuates based on in-game events. Changing environments are covered by scenarios such as a damaged base station (limited server cores), a weakened spaceship (limited client cores), or atmospheric anomalies (constrained network). These scenarios illustrate the real-world heterogeneity of data transfers across cloud, on-premise, and edge deployments.

Parameter Selection for Base Station and Spaceship: Before launching a mission, users configure data transfer parameters via the setup interface (see Figure 5a), which determines how efficiently data moves between the base station (server) and the spaceship (client). Each level introduces different constraints, such as limited


Environment characteristics

Base station server

Client space ship

```
Transfer Table:              ss13husallm
Table size:                  800 MB
Network bandwidth:           1000 MiB/s
Columns are 3 integers, 3 floats, 1 date, 1 string

Server system: csv                                Client system: csv
CPU cores: 8                                      CPU cores: 8
```

Global parameters

```
Network threads:             1
Buffer size (KiB):           64
Compression algorithm:       no_comp
Intermediate format:         row oriented
```

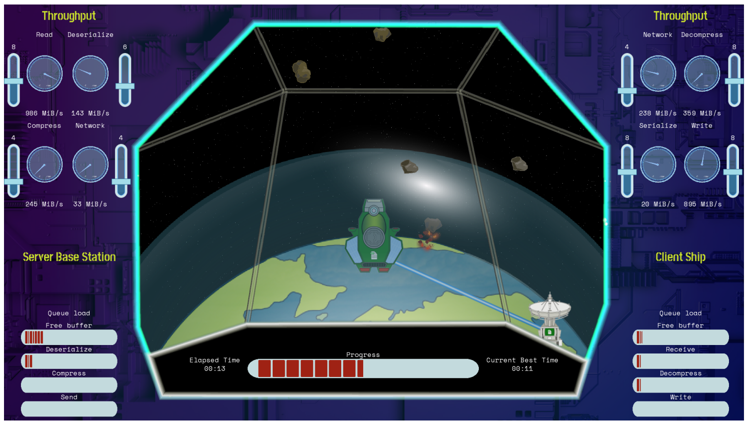
Server parameters

```
Read threads:                1
Read partitions:             1
Deserialization threads:     1
Compression threads:         1
Buffer pool size (KiB):      1024
```

Client parameters

```
Decompression threads:       1
Serialization threads:       1
Write threads:               1
Buffer pool size (KiB):      1024
```

(a) Parameter Selection.



(b) View from the cockpit: Users can change parameters and view performance impact.

Figure 5: Enter the Warp: Configuration screen and cockpit view, visualizing data transfer metrics and live reconfiguration.

CPU, restricted client resources, or high-latency networks, requiring informed decisions. For example, excessive parallelism can hurt performance in CPU-bound settings, while compression may add unnecessary overhead in high-bandwidth scenarios. Compression algorithms also trade higher compression ratios for greater compute cost, making them unsuitable for low-resource environments. To guide users, each level provides hints illustrating key trade-offs and linking decisions to real-world scenarios. Users must balance these parameters carefully to maximize transfer throughput.

Real-Time Transfers (Navigating the Warp): After selecting parameters, users *enter the warp*, initiating the data transfer mission (see Figure 5b). Inside the spaceship's cockpit, they monitor live system metrics, including per-component throughput and queue loads. This visualization shows how parameter choices impact performance, allowing users to identify bottlenecks and inefficiencies. Linking parameter changes to XDBC's internal metrics gives users insights into its dynamic adaptation and performance. Users can further refine strategy by adjusting parameters and observing their impact. In the background, an actual XDBC transfer runs: an optimized transfer strengthens Earth's shield, while inefficient setups let more meteors through. Users can directly observe how different environments and constraints influence throughput; for example, setting more threads than available cores may reduce performance due to contention, while enabling compression on a low-resource base station may hurt throughput but improve it in low-bandwidth, high-resource scenarios. After the mission, users add their name to the leaderboard and compete for top throughput.

Game Setup: We implemented our interface in `libGDX`, an open-source game developement framework based on `OpenGL`. The interface sends user configurations and interactions to a controller component, which in turn adjusts a docker-based simulation environment running actual XDBC transfers on the participating systems. The controller also collects the runtime metrics and forwards them to the frontend, to adjust the visualizations.

Takeaways and Engagement: Efficient data transfer is critical in modern applications. Our "Enter the Warp" game offers an interactive way to explore these challenges. By tuning parameters in a guided manner and observing real-time performance, users build intuition for optimizing data movement across diverse environments. The demo also showcases XDBC's runtime reconfiguration, addressing the challenge of finding good configurations in dynamic settings. A live leaderboard encourages competition, and top scorers will receive small prizes at the end of the sessions.

ACKNOWLEDGMENTS

We gratefully acknowledge funding from the German Federal Ministry of Education and Research under the grants BIFOLD25B and 01IS17052 (as part of the Software Campus project PolyDB). We thank Stefanie Lenk for designing the game assets for our demo.

REFERENCES

[1] 2025. PostgreSQL wiki | Foreign data wrappers. https://wiki.postgresql.org/wiki/Foreign_data_wrappers#Specific_SQL_Database_Wrappers. Accessed: July 15, 2025.

[2] 2025. Turbodbc - Turbocharged database access for data scientists. https://turbodbc.readthedocs.io/en/latest/. Accessed: July 15, 2025.

[3] Matthias Boehm. 2011. Cost-based optimization of integration flows. (2011). https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-67936

[4] Benedikt Didrich et al. 2025. Learning to Accelerate: Tuning Data Transfer Parameters. In *AIDB@VLDB 2025*.

[5] Haralampos Gavriilidis et al. 2023. In-Situ Cross-Database Query Processing. In *ICDE*. https://doi.org/10.1109/ICDE55515.2023.00214

[6] Haralampos Gavriilidis et al. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proc. ACM Manag. Data* 3, 3, Article 157 (June 2025). https://doi.org/10.1145/3725294

[7] Brandon Haynes, Alvin Cheung, and Magdalena Balazinska. 2016. PipeGen: Data pipe generator for hybrid analytics. In *SoCC*. https://doi.org/10.1145/2987550.2987567

[8] Mark Raasveldt and Hannes Mühleisen. 2017. Don't hold my data hostage: a case for client protocol redesign. *PVLDB* (2017). https://doi.org/10.14778/3115404.3115408

[9] Raghav Sethi et al. 2019. Presto: SQL on everything. In *ICDE*. https://doi.org/10.1109/ICDE.2019.00196

[10] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. 2005. Optimizing ETL Processes in Data Warehouses. In *ICDE*. https://doi.org/10.1109/ICDE.2005.103

[11] Alkis Simitsis, Kevin Wilkinson, and Petar Jovanovic. 2013. xPAD: a platform for analytic data flows. In *SIGMOD*. https://doi.org/10.1145/2463676.2465247

[12] Yuanyuan Tian et al. 2016. Building a Hybrid Warehouse: Efficient Joins between Data Stored in HDFS and Enterprise Warehouse. *ACM TODS* (2016). https://doi.org/10.1145/2972950

[13] Xiaoying Wang et al. 2022. ConnectorX: accelerating data loading from databases to dataframes. *PVLDB* 15, 11 (2022). https://doi.org/10.14778/3551793.3551847