Fast and Scalable Data Transfer Across Data Systems

HARALAMPOS GAVRIILIDIS, BIFOLD and Technische Universität Berlin, Germany
KAUSTUBH BEEDKAR, Indian Institute of Technology Delhi, India
MATTHIAS BOEHM, BIFOLD and Technische Universität Berlin, Germany
VOLKER MARKL, BIFOLD, Technische Universität Berlin, and DFKI GmbH, Germany

Fast and scalable data transfer is crucial in today's decentralized data ecosystems and data-driven applications. Example use cases include transferring data from operational systems to consolidated data warehouse environments, or from relational database systems to data lakes for exploratory data analysis or ML model training. Traditional data transfer approaches rely on efficient point-to-point connectors or general middleware with generic intermediate data representations. Physical environments (e.g., on-premise, cloud, or consumer nodes) also have become increasingly heterogeneous. Existing work still struggles to achieve both, fast and scalable data transfer as well as generality in terms of heterogeneous systems and environments. Hence, in this paper, we introduce a holistic data transfer framework. Our XDBC framework splits the data transfer pipeline into logical components and provides a wide variety of physical implementations for these components. This design allows a seamless integration of different systems as well as the automatic optimizations of data transfer configurations according to workload and environment characteristics. Our evaluation shows that XDBC outperforms state-of-the-art generic data transfer tools by up to 5×, while being on par with specialized approaches.

CCS Concepts: • **Information systems** → *Data federation tools*; **Extraction, transformation and loading**; **Data exchange**.

Additional Key Words and Phrases: Data Transfer, Data Loading Optimization, Data System Interoperability

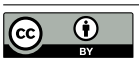
ACM Reference Format:
Haralampos Gavriilidis, Kaustubh Beedkar, Matthias Boehm, and Volker Markl. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 157 (June 2025), 28 pages. https://doi.org/10.1145/3725294

1 Introduction

The typical IT landscape of today's organizations comprises various decentralized data ecosystems and many data-driven applications [29, 34]. Over the last decades, there was a trend towards specialized data systems to improve performance and functionality [84, 85]. Apart from processing systems, modern data lakes allow storing raw data in various formats. Since data is distributed across these heterogeneous data representations and systems, efficient data transfer mechanisms are crucial for seamless integration and data analysis.

A Need for Data Transfer: There is a strong need for efficient data transfer because data movement and conversion is often resource-intensive and time-consuming. Accordingly, there is work on eliminating unnecessary data transfer. For example, in-database machine learning aims to conduct training and especially inference inside the DBMS [22, 49, 77] to avoid costly exports from the DBMS.

Authors' Contact Information: Haralampos Gavriilidis, BIFOLD and Technische Universität Berlin, Germany, gavriilidis@tu-berlin.de; Kaustubh Beedkar, Indian Institute of Technology Delhi, India, kbeedkar@cse.iitd.ac.in; Matthias Boehm, BIFOLD and Technische Universität Berlin, Germany, matthias.boehm@tu-berlin.de; Volker Markl, BIFOLD, Technische Universität Berlin, and DFKI GmbH, Germany, volker.markl@tu-berlin.de.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2836-6573/2025/6-ART157
https://doi.org/10.1145/3725294

Similarly, IoT systems push computation down to the edge to reduce unreliable network transfers [31, 92]. However, there are many use cases where data must be moved, either to leverage specialized tools, to integrate systems, or to consolidate data, e.g., data science applications [90], federated systems [26, 41, 44, 76, 78], extraction-transformation loading (ETL) flows [80], or data replication [56]. Overall, many modern data-driven applications would benefit from efficient data transfer mechanisms.

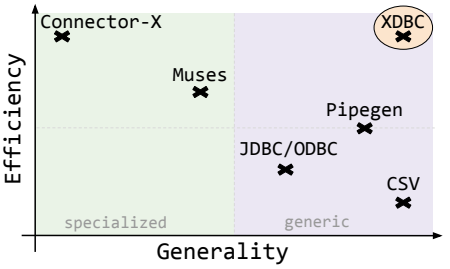


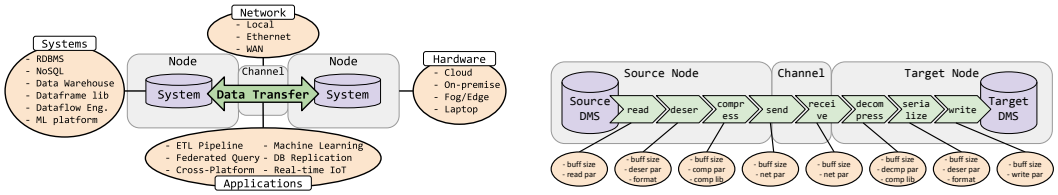
Fig. 1. Data Transfer Approaches.

Data Transfer Challenges: Designing an efficient data transfer mechanism is challenging due to the heterogeneity of data systems, particularly in terms of interfaces, data formats, and varying environment characteristics. Data transfer across heterogeneous systems is commonly realized through custom (i.e., specialized) point-to-point connectors, or generic solutions such as integration middleware or generic JDBC/ODBC drivers [6]. Figure 1 shows this trade-off space of efficiency versus generality. First, specialized connectors provide good performance but lack generality because they are implemented for specific pairs of systems. Second, generic adapters like JDBC/ODBC are primarily designed for lightweight DBMS access and suffer from poor performance [74]. Additionally, data transfer involves multiple steps (e.g., reading, deserializing, compressing, and transmitting), which must be tuned for data, system, and network characteristics. For example, compression may improve the transfer in unreliable consumer-cloud networks, but yield slowdowns in a cloud environment. The complexity of heterogeneous systems and environments renders the optimization of data transfer onerous. Hence, there is a need for a data transfer mechanism that yields robust performance, is easy to use, and general enough for data transfer across heterogeneous systems.

Limitations of Existing Work: Despite work on optimizing ETL [80, 81] and integration flows [28], the literature on efficient data transfer is relatively sparse. Raasveldt et al. analyzed DBMS client protocols and found that JDBC-based protocols are unsuitable for transferring large amounts of data due to their row-based nature and excessive metadata in each protocol buffer [74]. Pipegen generates pipes from existing CSV I/O unit tests in DBMSes, and redirects text I/O to compressed binary communication [48]. Zigzag joins optimize data transfers during joins of data warehouse tables and files in distributed file-system via Bloom filters [86]. Connector-X improves the loading of DBMS tables into pandas dataframes through parallelism and efficient data handling [90]. While these approaches optimize specific aspects of data transfer, there is a lack of a holistic data transfer framework that adapts to different physical environments and supports heterogeneous systems.

Contributions: To address the data transfer challenges and limitations of existing work, we introduce XDBC as a holistic data transfer framework. XDBC offers reader and writer interfaces that can be implemented for arbitrary data systems, eliminating the need for multiple connectors per system pair. XDBC's architecture comprises decoupled components (for read, deserialize, compress, send, receive, decompress, serialize, and write) as well as different physical implementations, which facilitate a flexible configuration and tuning for different environments and data characteristics. In order to free users from the need of manual tuning, XDBC's heuristic optimizer automatically selects effective configurations for given environments. Our detailed technical contributions are:

- **XDBC Architecture:** We describe the architecture and runtime of XDBC in Section 3. The simple yet very effective design allows seamlessly integrating existing data systems as well as tuning the individual data transfer components.



(a) Data Transfer Topologies and Applications. (b) Data Transfer Anatomy: Configurable Pipeline.

Fig. 2. Data Transfer Across Heterogeneous Systems

- **XDBC Optimizer:** We introduce a novel optimizer for automatically tuning the data transfer configurations in different environments in Section 4. This heuristic optimizer selects parallelization and transfer strategies, and adapts configurations to monitored runtime statistics.
- **Experiments:** Finally, we present the results of an extensive experimental evaluation in Section 5. We compare XDBC with a variety of baselines on real and synthetic datasets as well as different simulated network topologies.

2 Background

In this section, we introduce the necessary background of data transfer, guided by a running example. Data transfer is crucial for many applications, and existing solutions fall short for generic data transfers across heterogeneous systems in different environments.

2.1 Motivation and Running Example

As shown in Figure 2a, data transfer has many different dimensions: different applications run on different data systems and infrastructures (hardware and networking). In the following, we introduce a running example and describe the state-of-the-art data transfer.

Example 1 (Data Science Use Case). *Assume a data-science application, where data practitioners need to load a large dataset from a DBMS, such as PostgreSQL, into a data science framework, such as Python pandas, to pre-process the data, run exploratory data analysis, and train various ML models. The data transfer between the DBMS and pandas may occur in different environments with very different characteristics. First, pandas may run on a local laptop, connected via a slow wide-area-network to the cloud. Second, the data practitioner may run pandas in a remote, cloud-hosted, environment, such as Google Colab, which is better connected to the DBMS.*

Generic Data Transfer: To load the data from the DBMS into pandas, one can utilize generic ODBC-based connectors, such as the widely-used `turbodbc` [12] and `sql.reader` [8]. Such plug-and-play connectors are very practical: users install the drivers and issue queries against the DBMS. The row-based results are then put into a collection of objects such as a pandas dataframe. Most ODBC (and JDBC) drivers only offer limited tuning knobs to optimize the transfer. The most commonly used tuning parameter is the number of rows to fetch in each round-trip to the DBMS. To further optimize performance, users could manually issue multiple queries—that partition the table by primary key—and then assemble the partitions. Existing data loading tools (e.g., Connector-X [90] and Spark JDBC [11]) also follow this approach and generate the partitioning queries automatically. However, the used row format—ideal for row-oriented DBMSes—has shown to be inefficient due to metadata bloat of in-flight buffers and limited compressibility [74].

Specialized Data Transfer: An alternative solution is a custom (i.e., specialized) connector that bypasses the generic JDBC/ODBC driver and establishes a direct connection to the DBMS. This approach allows more fine-grained control and direct conversion from the DBMS-internal

format to the target dataframe, without expensive transformations through an intermediate format. For example, Connector-X [90], implements specialized connectors for different DBMSes and transforms their data directly to dataframes in parallel, while PostgreSQL specialized foreign data wrappers read data from other DBMSes [10]. The specialized connector approach—albeit enabling more optimizations—requires more engineering effort and is tailor-made for that specific pair of source and target systems. Moreover, adapting to the hardware and network environment requires manual tuning or additional substantial development effort.

2.2 Data Transfer Anatomy

Figure 2b shows data transfer as a streaming pipeline that involves a source and target system (potentially located in different environments) that communicate over a channel, i.e., network, memory, or file. We first extract data from the source system, and then transform them into an intermediate format (via deserialization). Subsequently, we may compress the data and then, send the data over a channel to the target. At the target node, we read the data from the channel, decompress the data if needed, transform the data into the target format (via serialization), and finally, load the data into the target system. While these steps could be individually configured, existing approaches tightly couple them.

3 XDBC Architecture and Runtime

Next, we introduce XDBC's architecture, individual components, and runtime techniques. We treat data transfer as a pipeline of configurable components, which allows for holistic optimization. To support extensibility for different pairs of data systems, XDBC offers reader and writer interfaces.

3.1 System Architecture

XDBC's high-level architecture is illustrated in Figure 3 and comprises a connector pipeline split across the *XDBC Server* and *XDBC Client*. The server pulls data from the source system and sends them to the client; the client writes data into the target system. Server and client execute a configuration from the *Optimizer*.

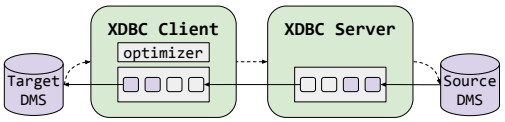


Fig. 3. XDBC: Gray modules provide core functionality, purple modules (read/write) are developer-provided.

Connectivity: Both server and client can run beside the source and target systems as a standalone process, or be directly embedded within them. For example, for our non-intrusive PostgreSQL XDBC connector, we run XDBC as a standalone process which communicates with PostgreSQL through its external API. For a deeper integration—similar to Raasveldt et. al [74] or Portage [39]—the XDBC server may also run as a PostgreSQL extension. Similarly, we implement the PostgreSQL client as a foreign data wrapper (extension), and the Apache Spark client as a custom data source. We bundle XDBC's core functionality as a shared library and offer the following extensibility mechanism:

- **XDBC Server:** Creating an XDBC server for a system requires implementing XDBC's reader and serializer. This interface defines methods to get memory from XDBC and push populated buffers back. Developers extract data using internal or external APIs, transform it to a supported intermediate format, and push it to XDBC's queues.
- **XDBC Client:** Creating an XDBC client is the opposite. We offer iterator-like methods that provide buffers in the intermediate format, which are transformed and loaded into the target system. For example, in our Spark client, we convert intermediate binary data to RDDs, whereas in our pandas client, we convert them to NumPy arrays and construct a dataframe.

Exposing core XDBC functionality as a shared library facilitates connectivity to data systems, enables broader integration, and keeps XDBC in control of the full data transfer pipeline.

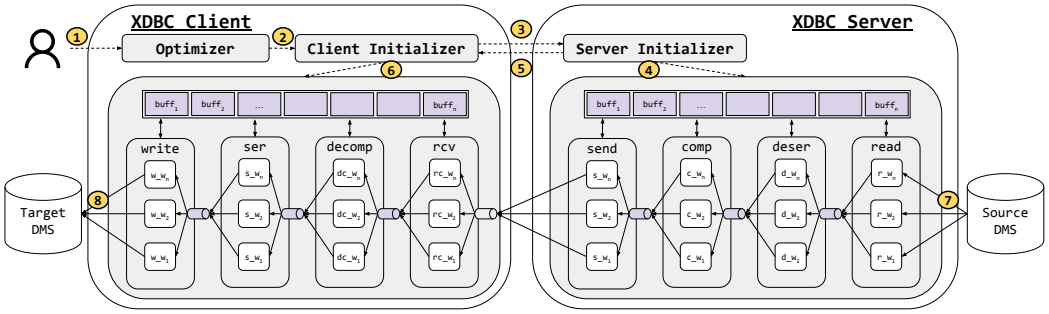


Fig. 4. XDBC Architecture & Flow: Users request data from the source system via the Client, which initializes the Server and starts the transfer. Dashed lines indicate control flow, solid lines indicate data flow.

Control and Data Plane: XDBC's architecture is illustrated in Figure 4 and supports the following tasks. First, users can submit data transfer jobs by providing the dataset, system access, and a configuration. Second, users can either manually specify the configuration, or use XDBC's optimizer (1), which we detail in Section 4. When starting a data transfer, XDBC initializes its pipeline with the given configuration (2). The client sends the configuration to the server (3) that initializes memory and components (4). Finally, the server signals (5) that the transfer starts (6). The data plane comprises the actual data transfer. Data enters the pipeline from the `reader` (7), which extracts data from the source system and copies them to internal buffers. Full buffers are forwarded to the `deserializer`, which transforms the data from the source format to the intermediate format. For instance, the PostgreSQL server transforms the PostgreSQL protocol format into our internal (row, columnar, or Arrow) format. The `deserializer` forwards the buffers to the `compressor`, which compresses the buffers and forwards them to the `sender`. Finally, the `sender` transmits the buffers over the network to the `receiver`. On the client side, the `receiver` copies the network buffers to the internal buffers and forwards these buffers to the `decompressor` which in turn forwards them to the `serializer` (for transforming the intermediate format to the target format). Finally, the `writer` loads the deserialized data into the target system (8). For example, in case of the Python pandas client, the `deserializer` creates the necessary Python structs from the incoming binary data and then the `writer` creates the final dataframe object. Overall, these components operate in a streaming fashion in order to leverage pipeline parallelism.

3.2 Runtime Techniques

Key runtime techniques—besides the individual data transfer pipeline components—are memory management and data-parallelism.

Memory Management: XDBC's memory manager allocates a user-defined fixed memory segment at startup and organizes it as a ring buffer (see Figure 5). This memory is divided into logical buffers (also user-defined) with unique IDs, each consisting of a header (of metadata like data size, record count, compression type, and format) and a payload (of the actual data), forming what we call the buffer pool.

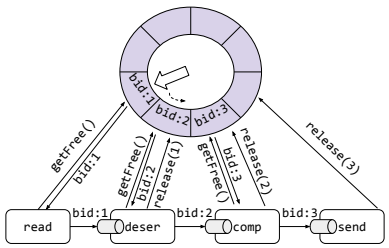


Fig. 5. XDBC Ring Buffer & Components.

Table 1. XDBC Components

Logical	Physical	Implementation
Read	CSV	custom reader
	Apache Parquet	`libarrow`, `libparquet`
	PostgreSQL	`libpq`
	MySQL	`libmysqlclient`
	Clickhouse	C++ client
Write	CSV	custom writer
	Apache Parquet	`libarrow`, `libparquet`
	PostgreSQL	fdw extension
	Apache Spark	custom data source
	Pandas	NumPy array creation
De-/Serialize	Row/Column	custom binary formats
	Apache Arrow	`libarrow`
	Apache Parquet	`libarrow`, `libparquet`
De-/Compress	no compression	C/C++ libraries
	zstd, snappy, lz4, lzo	
Send/Receive	TCP	boost.asio synchronous

The Client and Server processes maintain their own independent buffer pools. This buffer pool is shared among component workers. XDBC minimizes unnecessary memory movement between components by communicating buffer IDs rather than entire buffers. For example, in a typical pipeline configuration, the `reader` fetches a free buffer ID from the memory manager, fills it, and forwards the buffer ID to the `deserializer`. The `deserializer` retrieves the buffer ID from its incoming queue, fetches a free output buffer for the deserialized data, and, once finished, forwards the output buffer to the `compressor`'s queue while returning the input buffer to the memory allocator for reuse. This design reduces unnecessary memory allocation overhead.

Parallelism: XDBC combines pipeline parallelism, where different components operate sequentially, with data parallelism, where multiple component workers process buffers concurrently within each component. Each component has a shared input task queue for good load balance, and workers fetch buffer IDs from these queues, retrieve the buffers from the shared buffer pool, process them, and forward the resulting buffer IDs to the next component's queue. To manage termination, XDBC uses poison pills. The last active worker enqueues as many poison pills as there are downstream workers into the next component's task queue. When a worker receives a poison pill, it shuts down. Our design is similar to the QPipe [47] micro-engine, ensuring good instruction locality per thread and scalability across components and workers.

Handling Back Pressure: XDBC ensures that there are always enough free buffers for every worker in the buffer pool. We reserve buffers according to the total degree of parallelism across all components. In order to balance buffer availability across components, we use blocking queues with maximum queue capacities. When a queue reaches its capacity, enqueue operations block until space becomes available. For example, if the deserializer's input queue is full, the reader stops processing until the deserializer removes buffers from its queue. This mechanism not only prevents overloading components but also ensures that the pipeline operates at the pace of the slowest component, efficiently handling back pressure.

Listing 1. XDBC Client Interface

```
class XClient {
public:
    XClient(RuntimeEnv &xdbcenv);
    void initialize(const std::string &clientName);
    int startReceiving(const std::string &tableName);
    bool hasNext();
    buffWithId getBuffer();
    void markBufferAsRead(int buffId);
    void finalize();
};
```

3.3 XDBC Component Primitives

The logical pipeline components (read, deserialize, compress, send, receive, decompress, deserialize, write) can be instantiated through different physical implementations. In the following, we describe our builtin implementations as well as means of extensibility.

Existing Physical Implementations: Table 1 shows the builtin physical implementations of the logical components. In the following, we describe these groups of physical primitives in detail:

- **Read/Write:** XDBC supports different types of external systems (readers and writers), namely the DBMSes PostgreSQL, MySQL, Clickhouse; the CSV and Apache Parquet file formats; and the collections Spark RDDs and pandas dataframes. As an example, for PostgreSQL, we use the native client `libpq` to extract the data as text into our buffers. We open a connection and issue multiple parallel fetch queries, partitioned by tuple identifiers `ctid`. The PostgreSQL writer is embedded as a foreign data wrapper that populates a foreign table by serializing our internal binary intermediate format to PostgreSQL format. Other writers are Apache Spark (DataSource API, Java/Scala object creation through JNI and direct buffers), Python pandas (data frame of NumPy arrays, zero-copy of native arrays through pybind, explicit object conversion for strings/character types).
- **Serialization:** The deserializers and serializers convert system-specific data representations into our binary row- or column-based (or Arrow) formats, and vice versa. For example, the text-based PostgreSQL and CSV deserializers parse a buffer of delimiter-separated strings into the native types and place them into the row or column layout in the output buffer.
- **Compression:** For the compressor and decompressor, we use existing libraries (i.e., zstd, snappy, lz4, lzo). These libraries are called once per buffer with the pointer and length of this buffer as arguments. We then add the metadata to the buffer header. Compression and decompression use output buffers to avoid complex in-place operations.
- **Send/Receive:** For senders and receivers, we leverage the `boost.asio` library for efficient TCP transfers. Sender workers copy their input buffers into a `boost.asio` network buffer for sending the data over a TCP socket to the receiver. We use synchronous transfers, as the sender and receiver threads are only responsible for copying network buffers to internal buffers, and do not block the rest of the pipeline.

Extensibility: XDBC is highly extensible, enabling the integration of new systems, intermediate formats, and operators for compression and communication. To add a new system, one must extend the XDBC client and server (provided as shared libraries). Listing 1 shows the XDBC client interface. A deserializer and writer can be implemented with the shown methods. Specifically, the implementation initializes the runtime environment, starts the data transfer, and retrieves buffers using iterator-style methods. These buffers contain data in the intermediate format, which can then be deserialized and written to the target system. Similarly, readers can use the source

system's APIs to extract data into XDBC buffers, and deserializers for the intermediate format. The buffer-at-a-time processing facilitates a seamless integration of new implementations.

3.4 Intermediate Formats

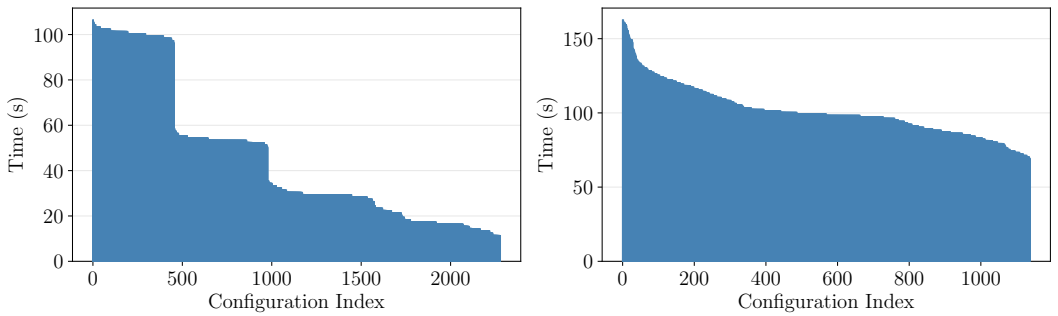
XDBC supports the following three intermediate formats, each with a schema (number of attributes, their sizes, and layout):

- **Row-based:** For the *row-based* format, we use the N-ary storage model, and write tuples consecutively to buffers using primitive data types. Accessing tuples involves calculating their offsets based on the tuple size (from schema) and index.
- **Columnar:** The *columnar* format has a PAX-like organization [23], where attribute arrays of all tuples are stored consecutively in buffers. This layout is favorable for systems with columnar storge. For example, when integrating Pandas, we copy columns directly into NumPy arrays by matching the schema-defined offsets. On the other hand, reconstructing full tuples from this format is more complex.
- **Apache Arrow:** We also support Arrow [14], a widely-used PAX-like format similar to our columnar format. We use Arrow's API to create `RecordBatches` (including metadata) and write those into our buffers, while appending metadata. Although Arrow's API has more overhead than our simpler columnar format, its adoption and built-in converters enable seamless integration with systems supporting it.

Extensibility: XDBC is agnostic to the intermediate format, which allows adding new formats through custom operators. To support a new format, one must implement the Deserialize and Serialize operators in the XDBC Client and Server interfaces, to transform data from the intermediate format to/from the system target/source format. For example, one could add an operator that transforms buffers with CSV content to the Apache Avro format.

4 XDBC Optimizer

The configurability of our data transfer framework allows for tuning and adaptation to different environments. Figure 6 shows the impact of picking the right parallelism configuration, with performance differences of up to 8×. However, since the configuration search space grows exponentially in the number of tuning knobs, manual tuning is difficult. In this section, we formulate the problem of automatically optimizing these data transfer configurations, and introduce a simple yet effective heuristic optimizer.



(a) Cloud-Cloud Environment (b) Edge-Fog Environment

Fig. 6. Impact of Tuned Data Transfer Configurations. Environment a) 16 cores for server/client, with unlimited network, and b) 8 / 2 cores for server/client, and 500MB/s network bandwidth. We randomly sample the parallelism degrees for server and client components and configure a fixed buffer size.

4.1 Problem Formulation

Our overall objective is to increase data transfer performance, in terms of throughput. In this section, we introduce the used notation, formulate the optimization objective and its constraints, as well as analyze the resulting search space of configurations.

Notation: Our data transfer framework can be viewed as a queueing system [30, 57], specifically as a cyclic customer model. In that model, we have customers rotating between servers, where customers are the entities that need to be processed and servers do the processing. We model our pipeline components as the servers, and the data buffers as the customers. Each server may have multiple workers assigned to it that resemble the parallel threads that process different customers. Whenever a customer (buffer) is done at a server (component), it moves to the subsequent server. Specifically, buffers are moved from the reader to the sender at the XDBC server, and from the receiver to the writer at the XDBC client. For modeling, we introduce the following notation:

- Queues (Q): A set of queues that represent the components (e.g., read, deserialize), denoted as $Q = \{q_1, q_2, \ldots, q_n\}$.
- Workers (w): Each queue q_i may have multiple workers w_i, where $|w_i|$ denotes the number of workers, and $\overline{W_s}$ and $\overline{W_c}$ are the maximum number of workers at the server and client.
- Configuration ($\mathbf{c}$): A vector of general (e.g., buffer size) and queue-dependent parameters (e.g., compression type).
- Service Rate (μ): The service rate $\mu_i(|w_i|, \mathbf{c})$ per worker, which essentially reflects the throughput measured in MB/s.

Optimization Objective: Our overall goal is to maximize the end-to-end throughput. To this end, we aim to find the optimal configuration $\mathbf{c}^*$—from the search space of all configurations $\mathcal{C}$—that minimizes the end-to-end transfer time (or equivalently, maximizes the service rate). We formally define the objective as follows:

$$\begin{aligned} \mathbf{c}^* &= \arg\max_{\mathbf{c}\in\mathcal{C}} \left(\min_{i\in[1,n]} \mu_i(|w_i|, \mathbf{c}) \right) \\ s.t. \sum_{i=1}^{n/2} |w_i| &\leq \overline{W_s} \ \wedge \sum_{i=n/2+1}^{n} |w_i| \leq \overline{W_c} \end{aligned} \tag{1}$$

Intuitively, in a streaming pipeline, the minimum service rate of a component determines the end-to-end service rate. For example, if the pipeline is network bound, there is no benefit in increasing the degree of parallelism of other components. The available tuning knobs (parameters that affect performance) of configurations are summarized in Table 2. Equation (1) finds the configuration that maximizes the end-to-end service rate under the constraint that the assigned degree of parallelism at the server and client do not exceed the maximum numbers of workers $\overline{W_s}$ and $\overline{W_c}$.

Estimated Throughput: Furthermore, every component has a pre-determined base throughput $\mu_i(1, \mathbf{c})$ (single-threaded processing rate in MB/s), which we scale sub-linearly—in order to reflect realistic scaling of data-intensive operations—for a given degree of parallelism $|w_i|$ as follows:

$$\mu_i(|w_i|, \mathbf{c}) = (s + (1 - s) \cdot |w_i|) \cdot \mu_i(1, \mathbf{c}) \tag{2}$$

This scaling follows Gustafson's law [46], where s denotes the serial fraction of a component; yielding, for example, a speedup of 28.9× for 32 workers and a serial fraction of $s = 0.1$. For compression libraries, we also estimate the compression ratios according to chosen buffer sizes (the larger, the better the ratio) and intermediate formats (better compression for columnar storage), which in turn, influences the effective service rates of senders and receivers. The base throughput of different components as well as compression ratios of compression libraries are obtained via

Table 2. Component Configuration Params and Values.

Configuration	Parameter	Example Values
General	buffer pool size (KiB)	2048, ..., 32768
	buffer size (KiB)	4, ..., 1024
	format	row, column
	compression lib	none, zstd, snappy, lzo, lz4
Server Workers	read	[1, max workers server]
	deserialization	[1, max workers server]
	compression	[1, max workers server]
	send	[1, max workers server]
Client Workers	receive	[1, max workers client]
	decompression	[1, max workers client]
	serialization	[1, max workers client]
	write	[1, max workers client]

offline profiling over a variety of datasets. By default, the XDBC optimizer uses these pre-packaged statistics for ranking plans to simplify the out-of-the-box deployment. However, users can explicitly trigger this profiling for their systems as well as data and environment characteristics in order to improve the accuracy of the optimizers cost model.

Constraints: Our optimizer respects the physical limits of the environment. First, we ensure that the number of client/server workers do not exceed the available degree of parallelism. Second, the nominal I/O bandwidth (of memory or storage) as well as network bandwidth determine upper bounds for the reachable read/write and send/receive service rates. According to Equation (1), the maximum reachable end-to-end service rate μ_{max} is then upper bounded by the minimum upper bound of these components.

Search Space: The tuning knobs of Table 2 create an 12-dimensional hypercube of possible configurations. The number of configurations is given by $\mathcal{O}(\prod_{j=1}^{12} d_j)$ where d_j is the domain (number of distinct items) per parameter. However, a large fraction of these configurations are invalid, e.g., when violating $\sum_{i=1}^{n/2} |w_i| \leq \overline{W_s}$. For example, assuming $\overline{W_s} = 16$ and four server components: of all $16^4 = 65{,}536$ configurations only 1,820 are valid. Nevertheless, with all knobs and large parameter domains, full enumeration is impractical and hence, we devise a heuristic but very fast optimizer.

4.2 Optimization Algorithm

Our heuristic optimizer aims to find a good—but not necessarily optimal—data transfer configuration for the given environment and dataset. Here, we describe our basic heuristic approach, optimization algorithm, and extensions for inter/intra-transfer adaptation.

Heuristic Approach: The design of our heuristic optimizer deals with four major categories of parameters and rewrites:

(1) Parallelism (major tuning knob of client/server components)
(2) Compression (increases network throughput, adds overhead)
(3) Memory management (influences resource consumption and compression, depends on parallelism)
(4) Additional rewrite rules (e.g., bypass deserialization)

Algorithm 1 Heuristic Data Transfer Optimizer

Input:
  $\mu_i(1, \mathbf{c})$: base throughput for each queue q_i
  $\mu_{max,i}$: maximum throughput for each queue q_i
  $\mathbf{s}$: speedup-related serial fractions
  $\overline{W_s}, \overline{W_c}$: max numbers of server and client workers
Output:
  $\mathbf{c}^*$: found configuration
  $estimated\_throughput$: estimated total throughput
// Step 1: Minimum Upper Bound
$\mu_{max} \leftarrow \min(\mu_{max}(q_i) \forall i)$
// Step 2: Initialization
for each i in q **do**
  $\mathbf{c}_i \leftarrow 1$
end for
// Step 3: Iterative Search for Optimal Workers
repeat
  Identify the two slowest queues: q_i, q_j based on $\mu_i(|w_i|, \mathbf{c})$
  while $\mu_i(|w_i|, \mathbf{c}) \leq \mu_j(|w_j|, \mathbf{c})$ **or** $\sum_{i=1}^{n} |w_i| \leq \overline{W}$ **do**
    $|w_i| \leftarrow |w_i| + 1$
    Update the service rate $\mu_i(|w_i|, \mathbf{c})$
  end while
until optimal distribution or maximum workers reached
// Step 4: Compression Pass
if network is the bottleneck **then**
  Re-optimize by re-running Step 3 with compression in $\mathbf{c}$
end if
return $\mathbf{c}^*$, $estimated\_throughput$

We apply these rules in order of their anticipated impact on performance, based on experimental analysis. First, we iteratively assign all available workers to the different components according to their service rates. In every iteration, we increase the parallelism of the slowest component. Second, we decide if compression might be beneficial by comparing the network throughput with the min-max service rates of other components and the so-far best estimated throughput. If communication is the bottleneck, and there are still available workers, we rerun our iterative assignment by considering all compression algorithms (according to estimated compression ratio and compression/decompression throughput). Third, we decide the buffer size, number of buffers, and format. We set the buffer size according to L1 cache sizes (typically 32KB), use a fixed number of buffers per worker to avoid over- or under-provisioning, and use columnar formats if compression is enabled or the source/target systems also have columnar representations. Fourth, there are additional rules to bypass de/serialization operators when the source and target systems are the same, or when data can be serialized directly from the source format. If the source data is compressed (e.g., Parquet) and the serializer can handle transformations (e.g., Parquet to Pandas), we transfer the compressed data and perform decompression and deserialization at the client. Focusing on these parameters categories separately simplifies the related rewrites and transformation rules while still yielding good configurations.

Algorithm Description: Algorithm 1 shows our heuristic optimization algorithm, which takes the constraints and base service rates as input (Lines 2-5) and returns the found data transfer configuration (Line 27). First, we compute the minimum upper bound service rate in Line 10, and initialize a working configuration **c** with one worker per component in Line 13. In an iterative fashion, we then pick the two slowest components and incrementally add workers to the slowest until it is no longer the slowest, and repeat this process until convergence (Lines 16-22). With the pre-final configuration (with assigned parallelism), we then evaluate if compression is amenable and repeat the reassignment of parallelism with enabled compression in Line 25.

Example 2 (Optimizing Parquet-PostgreSQL Transfers). *Assume transferring a collection of columnar Parquet files to a row-based PostgreSQL over a 1 Gbit network (125 MB/s peak transfer). Since the source is columnar, we heuristically pick our columnar intermediate format. Columnar formats show better compressibility; but column to row conversion is expensive, so if the client is weak, the server should convert it. Furthermore, assume the following base throughput of read, deserialize, compress, send, receive, decompress, serialize, write:* $[1000, 100, 0, 125, 200, 0, 110, 250$ *MB/s*$]$. *We start distributing parallelism across components, starting with the deserializer (the slowest). With* $|w_i| = 2$ *the deserializer is at 190 MB/s and the serializer at 209 MB/s and thus, already surpassed the upper bound. Due to the networking bottleneck, we now evaluate compression. With an estimated compression ratio of 3x, we get a new networking throughput of 375 MB/s and distribute remaining parallelism, starting with the deserializer, so that all components run at slightly over 375 MB/s.*

Algorithm Analysis: The described algorithm finds good configurations with very low overhead. The assignment in Lines 16-22 only requires at most $\mathcal{O}(\overline{W_s} + \overline{W_c})$ steps and is repeated at most $\mathcal{O}(n)$ times. Since the maximum number of workers $\overline{W_s}$, $\overline{W_C}$ and pipeline length n are very small its runtime is negligible. Interesting future work includes exact optimization (e.g., via an ILP formulation) with dedicated pruning strategies of invalid configurations.

Adaptive Optimization: By default, our optimizer utilizes average statistics—based on offline profiling over multiple datasets—for component service rates and compression ratios. For larger data transfers, we could also run the pipeline on a small sample of buffers for more accurate statistics before optimization. Interesting future work (which we did not implement yet) includes inter- and intra-transfer adaptation. First, inter-transfer adaptation starts from our pre-packaged average statistics, updates these statistics as moving averages with actual measurements in the deployed environment and systems, and utilizes these statistics for optimizing future data transfers. Second, intra-transfer optimization could—similar to intra-query re-optimization [35]—update the statistics and the configuration of a running data transfer in-flight, which allows better adaptation to data characteristics.

5 Experiments

Our experiments study XDBC—in comparison with state-of-the-art approaches for data transfer—in several applications. We further evaluate the impact of XDBC's configuration parameters as well as the XDBC optimizer in different environments. Overall, we find that XDBC yields substantial improvements compared to generic baselines and is competitive even with specialized connectors.

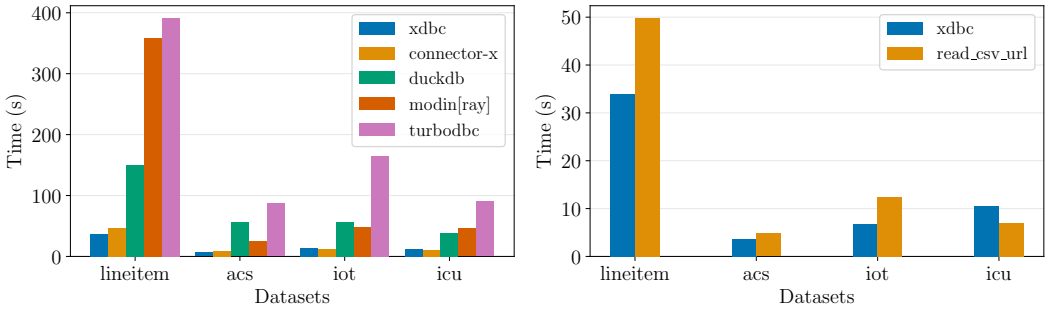
5.1 Experimental Setup

In the following, we describe our experimental setup.

Hardware/Software: We perform our experiments on a machine with an Intel(R) Xeon(R) Silver 4216 CPU @ 2.1 GHz and 16 cores (and 2 threads per core), 512 GB of main memory, and a 2 TB SSD. Furthermore, we use Ubuntu v20.04.2 LTS as an operating system running Docker v27.1.1. For simulating realistic environments, we use Docker resource constraints [17] to control the CPU cores and `docker-tc` [18] to control the network interfaces. For our DBMS experiments we use

Table 3. Datasets and their Data Characteristics.

Dataset	Size	#Rows	#Cols	Data Types
Lineitem [20]	7.2 GB	60M	16	int:4, double:4, string:8
US Survey [13]	1.2 GB	1.5M	230	int:230
IoT events [16]	3.3 GB	5.8M	85	int:39, double:41, string:5
ICU events [53]	2.2 GB	8.0M	26	int:9, double:6, string:11



(a) Pandas from PostgreSQL (b) Pandas from CSV

Fig. 7. Data Science Experiment (Cloud).

PostgreSQL server v13. For our data science experiments, we use Python v3.9 with pybind11 v2.13.6. For our dataflow engine experiments, we use Spark v3.3.1 on Scala v2.12.15 and sbt-jni v1.5.4.

Baselines: We implement XDBC in C++ and provide it as a shared library. To build our python bindings, we use pybind11, and for our Scala bindings sbt-jni. For our data science baselines, we use pandas v2.2, duckdb v1.0.0, sqlalchemy v2.0.35, and turbodbc v4.4.0. For our ETL baselines, we use Apache Spark v3.3 with PostgreSQL's JDBC driver v42.7.4 and PostgreSQL's native pg_fdw and PGSpider's jdbc_fdw [19] v0.4.0. Additional specialized baselines include ConnectorX [90] v0.3.3 and Modin on Ray [67, 73] v0.30.1.

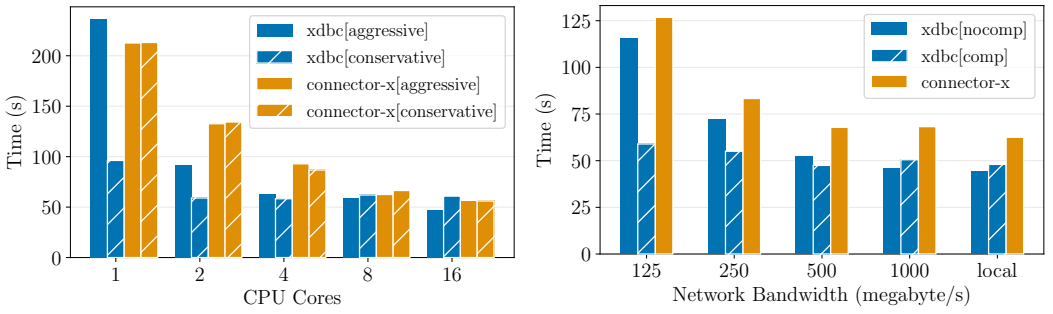
Datasets and Use Cases: Table 3 summarizes the characteristics of the used datasets. We selected these datasets because of their use in related work [74, 90] as well as their diverse characteristics (number of rows and columns) and data types. Our use cases comprise data science (PostgreSQL to pandas, and CSV to pandas) and ETL (PostgreSQL to Spark, and PostgreSQL to PostgreSQL).

5.2 End-to-End Data Transfer

We first present XDBC's end-to-end evaluation across various applications and environments.

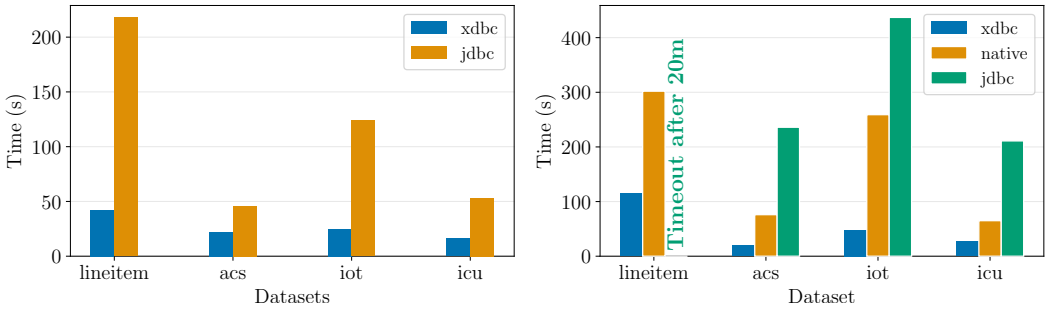
Data Science Use Cases: Many data science applications require moving data from a data store (e.g. an object store or an RDBMS) to a data science environment, e.g. Python pandas. Figure 7 shows the results of XDBC and state-of-the-art baselines. First, Figure 7a shows the results for loading data from PostgreSQL into a pandas dataframe. XDBC substantially outperforms the generic turbodbc (by an order of magnitude), DuckDB, and Modin, while also matching and sometimes improving ($\approx$ 15%) the runtime of the specialized Connector-X. Second, Figure 7b shows the results for transferring a remote CSV file to pandas. The baseline uses pandas' builtin `read_csv` with PyArrow. We see that for the largest dataset (lineitem), XDBC is $\approx$ 1.5$\times$ faster than the default HTTP CSV reader, while for smaller datasets the difference becomes negligible.

Insight: XDBC shows performance competitive with specialized data transfer techniques like Connector-X and PyArrow.



(a) Scaling the Client (CPU) (b) Scaling the Client (Network)

Fig. 8. Data Science: Scaling the Environment.



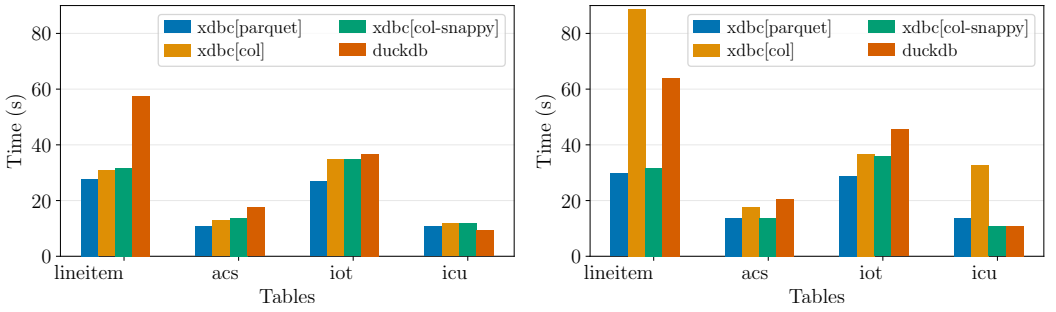
(a) PostgreSQL to Spark (b) PostgreSQL to PostgreSQL

Fig. 9. ETL/Database Migration.

Data Science in Varying Environments: To evaluate connector behavior across environments (e.g., cloud-to-laptop transfers), we vary CPU cores and network bandwidth (Figure 8a). XDBC uses two client configurations: `conservative` (1 worker/component) and `aggressive` (ser: 8, decomp: 4, snappy), and a default server (read: 10, deser: 4, decomp: 4). Connector-X also uses `aggressive` (8 threads) and `conservative` (matching available cores). For many cores (8, 16), both perform similarly, but with fewer cores (1, 2, 4), XDBC outperforms Connector-X by up to 2×, as Connector-X struggles to parallelize PostgreSQL reads effectively on limited cores. However, `xdbc[aggressive]` underperforms with 1 thread due to overprovisioning but becomes competitive at 8. Figure 8b shows the results with varying network bandwidth at 8, where `xdbc[comp]` and `xdbc[nocomp]` use the same `[aggressive]` configuration, but with and without snappy compression, respectively. We observe that beyond 250 MB/s, network is no longer the bottleneck and thus, `xdbc[nocomp]`, `xdbc[comp]`, and Connector-X show similar runtime. However, at 125 MB/s (e.g., 1 Gb Ethernet), network bandwidth becomes the limiting factor, and thus, compression in `xdbc[conf]` pays off with a 2× improvement over `xdbc[nocomp]` and 2.5× over Connector-X. For large network bandwidths, compression causes unnecessary overhead, and thus `xdbc[nocomp]` performs better.

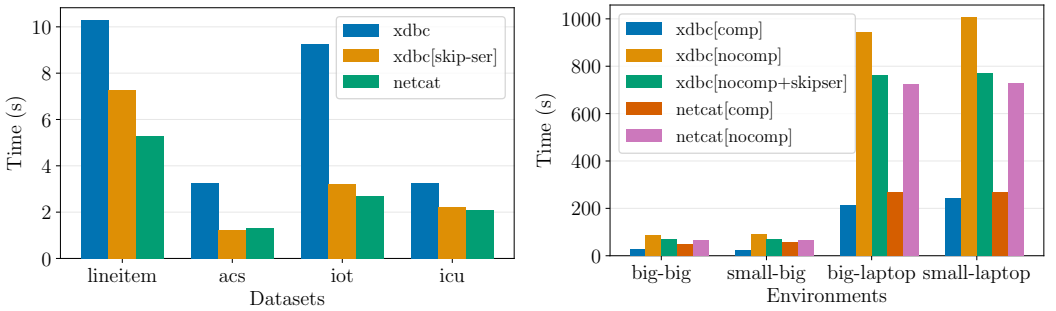
Insight: Picking the right parallelization and compression strategies allows XDBC to achieve robust performance even in constrained environments with limited resources.

ETL Use Cases: ETL use cases often require consolidating data from operational source systems into data lakes or data warehouses. To emulate these use cases, we conduct experiments for loading a table from PostgreSQL into Apache Spark and PostgreSQL, respectively. First, Figure 9a shows the results for bringing data into Apache Spark. Similar to the data science use cases, XDBC outperforms JDBC (parallelism degree of 8) for the lineitem dataset by 4×, and for the IoT dataset by 5×. These



(a) Unrestricted Network (b) 125 MB/s Network

Fig. 10. ETL: Parquet to CSV

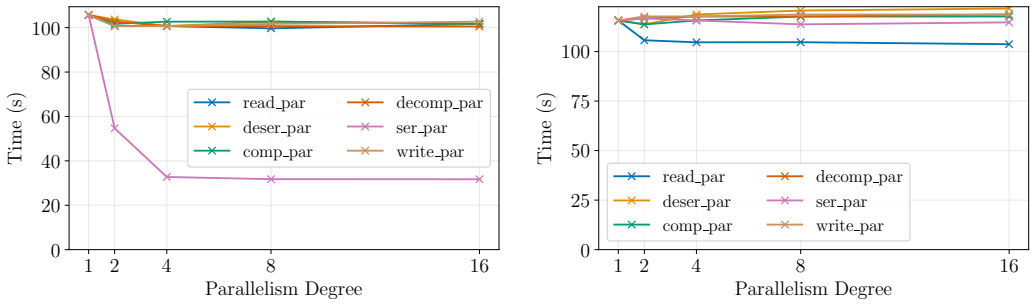


(a) Local Copy (b) Distrib. Topology (lineitem)

Fig. 11. Cloud: CSV to CSV in Different Environments.

results show the inefficiency of existing generic JDBC-style connectors, as they were designed to fetch little data for OLTP applications, as well as the performance impact of integrating XDBC with widely used data processing systems. Second, Figure 9b shows the results for PostgreSQL to PostgreSQL data transfers and thus, is representative for both data warehouse environments as well as federated data processing and database migrations. In this case, we implement a PostgreSQL foreign data wrapper around XDBC, and compare it to the PostgreSQL native foreign data wrapper (`pg_fdw`), and a JDBC data wrapper (`jdbc_fdw`). XDBC outperforms `jdbc_fdw` by more than an order of magnitude, and even the native `pg_fdw` by 2−4×. We attribute the performance difference to PostgreSQL sequentially serializing, reading, and writing data independent of the global parallelism configuration, while XDBC parallelizes the data transfer. Third, Figure 10 shows the results for a Parquet to CSV data transfer in two common ETL environments. We examine three XDBC configurations: (1) Parquet as the intermediate format with CSV conversion at the Client, (2) our columnar format with CSV conversion at the server, and (3) our columnar format with compression. The DuckDB baseline uses its remote Parquet reader. Using Parquet is beneficial in the unrestricted network environment because only one conversion is necessary (Parquet to CSV), while using our columnar format leads to slightly worse performance because of the two conversions (Parquet to columnar and columnar to CSV) and ≈ 70% more data transfer. XDBC outperforms DuckDB because of the parallel conversion for the larger datasets. In the restricted network environment—with 125 MB/s, where data transfer is the bottleneck—Parquet shows the best performance, but our columnar format with snappy compression is competitive because of its reduced data transfer size.

Insight: XDBC shows substantial performance improvements compared to generic and specialized solutions in data warehouses.



(a) CSV to CSV (b) PostgreSQL to CSV

Fig. 12. Cloud: Impact of Parallelism (default 1).

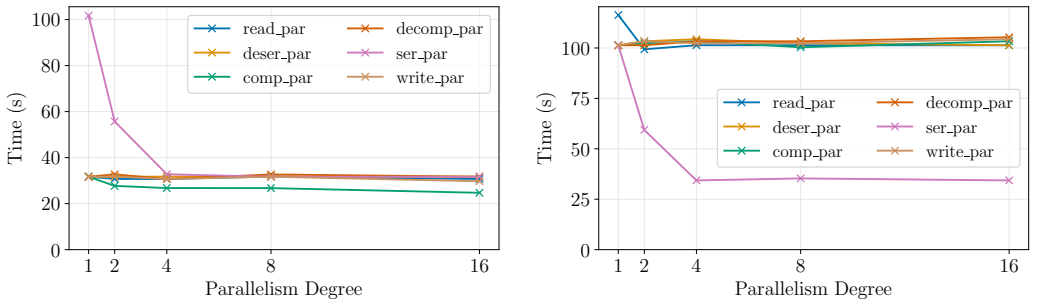
Cloud Use Cases: Disaggregated storage in the cloud enables independent scaling of compute and storage but requires data transfers across nodes and data centers. To evaluate XDBC's overhead, we compare it with simple file copies using `netcat`. In local environments (Figure 11a), XDBC is up to 2× slower than `netcat` for the largest dataset (lineitem). However, applying XDBC's skip serialization rule (bypassing deserialization and serialization) reduces this to 1.4×, with comparable performance to `netcat` for smaller datasets. Unlike `netcat`, which transfers raw byte streams, XDBC executes a generic pipeline—read, deserialize, send, receive, serialize, and write—designed for heterogeneous system environments. To validate the findings of our Docker-simulated environments, we conducted additional experiments on three physical nodes (big: same as Section 5.1, small: 4-core Intel (R) Xeon (R) E5530 at 2.40Ghz, 50GB of memory, and laptop: 14-core Intel (R) Core (R) i7-1370P, 64GB of memory) as well as two different networks using a 1 Gb Ethernet (125 MB/s) and a 200 Mb WAN (25 MB/s). Big and small servers are connected via 1 Gb Ethernet and the laptop via WAN. The results in Figure 11b are consistent with result in simulated environments (Figure 11a, 125 MB/s not shown): `netcat` outperforms XDBC without compression and in network-bound environments, XDBC with compression outperforms `netcat`, even with multi-threaded de/compression applied before and after the transfer.

Insight: XDBC shows performance close to raw byte-stream copies in local environments due to pipeline parallelism, and can improve performance for slow networks due to compression.

5.3 Micro Benchmarks

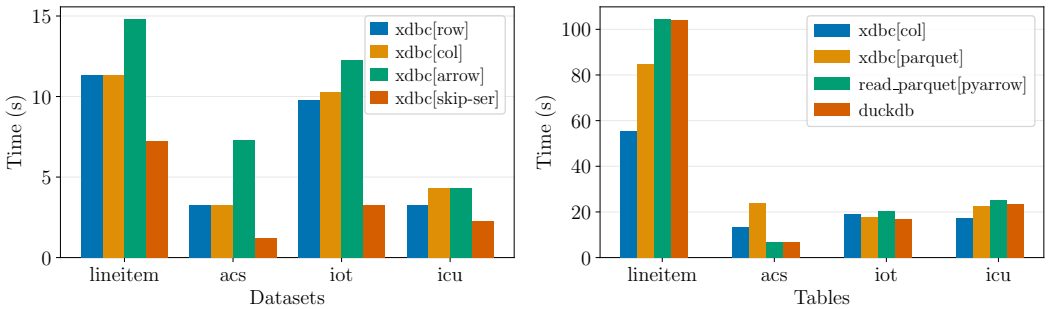
To study the individual XDBC components and their parameters in different environments, we perform a series of micro benchmarks. In order to avoid a combinatorial explosion, we present selected experiments with the following defaults: transfer of in-memory CSV files, a parallelism degree of 1, the row format, a 40 MB buffer pool, and a 1 MB buffer size. We then explore the parameters independently and report the mean of 10 repetitions.

Impact of Parallelism: We first investigate the impact of parallelization, i.e., number of workers for read, de/serialize, de/compress, and write. Figure 12 shows the results for transferring the lineitem table for CSV to CSV and PostgreSQL to CSV in a cloud environment (16 cores for server & client, unrestricted network), where we vary the parallelism of individual components, using snappy compression. For the CSV source, scaling the serialize parallelism significantly improves the performance (8 workers, see Figure 12a) because the serialization component dominates the runtime for local transfers. In contrast, for the PostgreSQL source, increasing the degree of parallelism for the reader improves performance (see Figure 12b). For the rest of the components, increasing the parallelism slightly degrades performance. The pipeline performance is bound by the slowest component,



(a) CSV to CSV (ser 8) (b) PostgreSQL to CSV (read 8)

Fig. 13. Cloud: Impact of Parallelism (default 1).



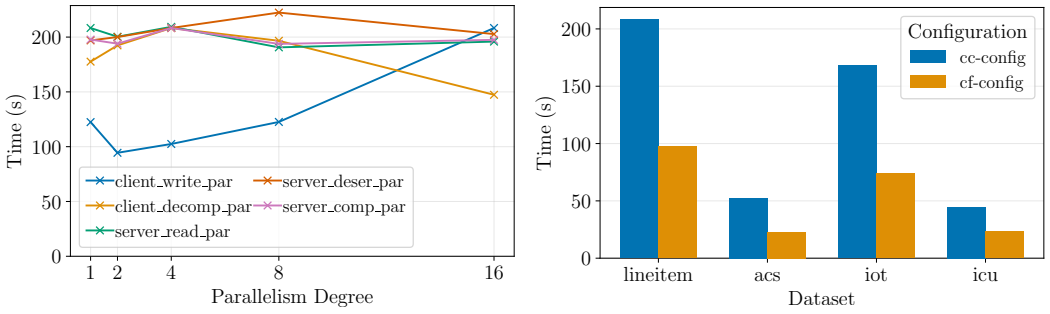
(a) Custom vs. Apache Arrow (b) Pandas from Parquet

Fig. 14. Comparing Intermediate Formats.

and thus, increasing the parallelism of others does not yield improvements. In order to find the second dominating component, we set the serialization parallelism to 8 for CSV to CSV and the read parallelism to 8 for PostgreSQL to CSV, keep the rest of the parameters at 1, and then again vary the parallelism of individual components. For CSV to CSV (Figure 13a), we see that increasing compression parallelism improves performance marginally, whereas for PostgreSQL to CSV (Figure 13b) increasing the serialization parallelism yields a substantial runtime improvement of up to 3×. In conclusion, different systems need different configurations, and parameters cannot be tuned independently of each other due to pipeline parallelism where the slowest component determines throughput.

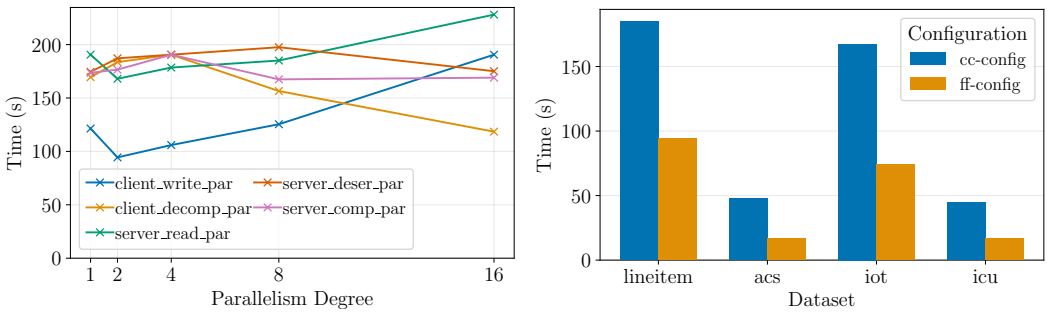
Insight: Parallelism tuning requires holistically tuning parameters to address pipeline bottlenecks and optimize performance.

Impact of Intermediate Formats: Figure 14a compares different intermediate formats for a CSV to CSV transfer, as well as plain copies without de-serialization. De-serialization incurs ≈ 30% overhead for the largest dataset, and Arrow exhibits more overhead than our row and column formats, which is likely due to Arrow's API which involves several function calls. Figure 14b shows the results for loading data from remote Parquet files to pandas. We compare two XDBC intermediate formats: Parquet (compressed) and our columnar format. The baselines are the builtin HTTP Parquet reader with PyArrow, and DuckDB, which has its own Parquet reader. We observe that XDBC with the columnar format outperforms the baselines for the datasets containing strings, as string conversion is expensive. XDBC's pandas connector converts the strings in parallel on receive. Parquet transfers ≈ 70% less data but the builtin Parquet converters show poor performance.



(a) Scaling Parameters Impact (b) Different Configurations

Fig. 15. Cloud to Fog: Impact of Parameters and Datasets.



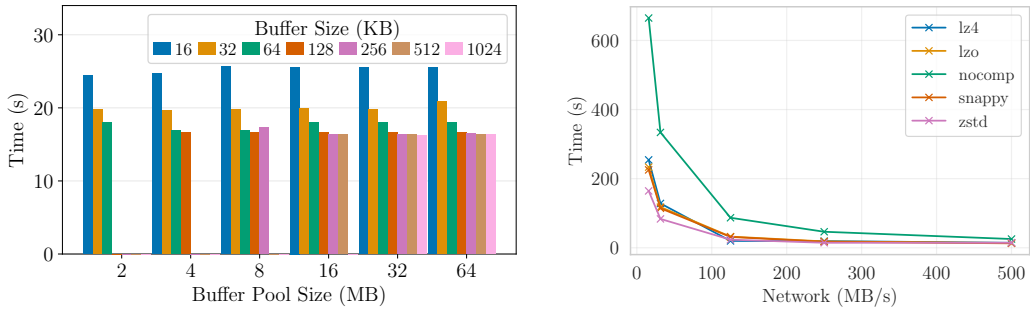
(a) Scaling Parameters Impact (b) Different Configurations

Fig. 16. Fog to Fog: Impact of Parameters and Datasets.

Insight: Format conversion is inevitable when source and target formats differ; parallelizing deserialization and serialization along with other steps mitigates such conversion costs.

Impact of Environments: A single default configuration cannot yield good performance in all environments. To quantify this effect, we conduct experiments in cloud-cloud, cloud-fog, and fog-fog environments. First, we use the best configuration from our cloud-cloud environment (read:1, deser:4, comp:snappy:4, decomp:4, write:16) and run the data transfer in cloud-fog. After having fixed the configuration, we scale the individual parameters in Figure 15a. We observe that the parameter with the most impact is the write parallelism (best performance is achieved with a write parallelism of 2, further increasing the parallelism degrades the performance). For decompression parallelism, performance degrades from 1 to 4 threads but then improves for 8 and 16 threads, where it is 1.2× better than with 1 thread. In contrast, scaling read and compress does not impact the performance, while the deserialization gets marginally worse when increasing the parallelism, and improves at 16 threads. These non-monotonic behaviors are due to complex interactions of queue load balancing, over-provisioning, and environmental factors, such as memory I/O. Furthermore, we compare the best cloud-cloud configuration (cc-config) with the best cloud-fog configuration (cf-config) in cloud-fog (see Figure 15b). For all datasets, cf-config yields a 2×+ performance improvement. Figure 16 shows the results of repeating these experiments in a fog-fog environment, where we see very similar characteristics. Thus, tuning configurations for the environment is of utmost importance.

Insight: Environments require tailored configurations; the best configuration in one environment may underperform in others.



(a) Buffer vs. Buffer Pool (b) Network vs. Compression

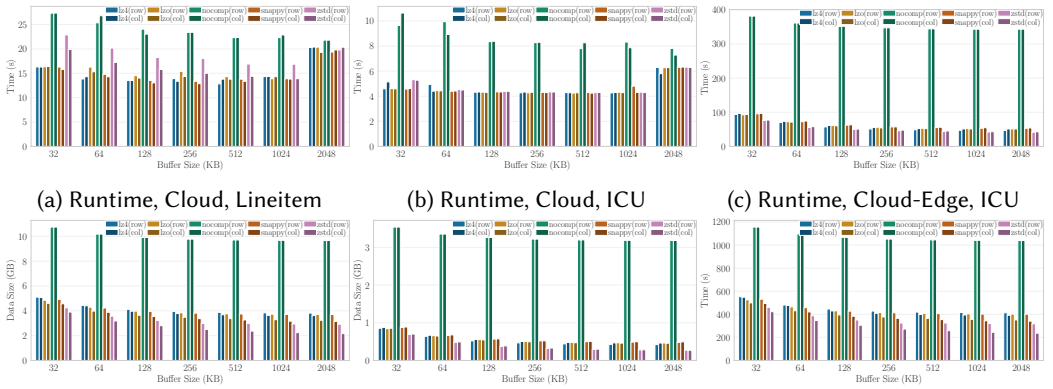
Fig. 17. Scaling Environment/Parameters.

Impact of Memory Management XDBC operates on a fixed memory budget, split into buffers (see Section 3.2). Larger buffers improve compression and transfer speed but reduce in-flight buffers, limiting pipeline parallelism. Smaller buffers increase thread communication, causing queue contention. To explore this trade-off, we scaled these parameters (XDBC conf. read/write:1, de/ser:8, de/comp:2, snappy without skipping deserialization) in a cloud environment (1000 MB/s). Our experiment results in Figure 17a show that i) smaller buffers worsen performance due to increased communication and lower compressibility and hence more data transfer, and ii) XDBC can efficiently transfer GBs of data with just 2MB of memory. Note that some buffer sizes are infeasible due to insufficient in-flight buffers required by XDBC (see Section 3.2).

Insight: Effective memory management depends on tuning buffer pool and buffer sizes to match the workload. With the right configuration, high-performance data transfer is possible even with limited memory budgets (2MB).

Impact of Compression: Compression can yield runtime improvements, especially in restricted networks. However, compression may also add overhead if network is not the bottleneck. Furthermore, the buffer size and intermediate format influences the performance of compression, both w.r.t. runtime and w.r.t. compression ratio. Additionally, parallelism may help mitigate compression overhead. To better understand these trade-offs, we conducted a series of micro experiments with different buffer sizes, compression libraries, and degrees of parallelisms in different environments:

(1) **Compression Effectiveness:** To understand the impact of compression, we conducted an experiment with different compression libraries in different network environments. The results in Figure 17b show that for up to 100MB/s the heavyweight zstd is more beneficial, while from then on other more lightweight compression mechanism offer slightly better performance, and at 500MB/s compression does not improve performance.
(2) **Buffer Size and Format:** Figure 18a shows the runtime for transferring the lineitem dataset in an environment with 16 client/server cores and 1000 MB/s network. Here, compression can yield a speedup of 2×. For small buffer sizes of 64 and 128KB, the compression libraries perform similarly and zstd is the slowest (up to 1.8×). For larger buffer sizes through, the intermediate format plays a role, i.e., a columnar format can yield a runtime reduction of ≈ 15%, compared to the row format. For a very large buffer size of 1MB, compression slows down due to larger dictionaries. Figure 18b shows similar results for the ICU events dataset. However, for this dataset, the columnar format is not beneficial. Overall, in this cloud environment, a buffer size of 256-512 KB, a lightweight compression library (e.g., snappy/lz4), and columnar format are desirable.



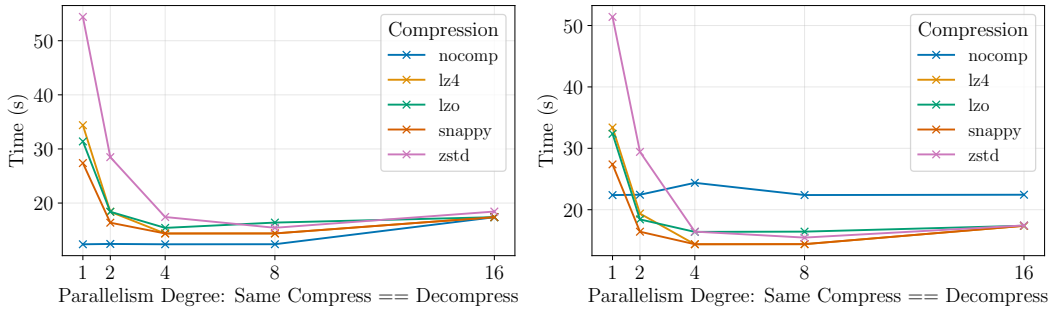
(a) Runtime, Cloud, Lineitem (b) Runtime, Cloud, ICU (c) Runtime, Cloud-Edge, ICU

(d) Transferred Size, Cloud, Lineitem (e) Transferred Size, Cloud, ICU (f) Runtime, Cloud-Edge, Lineitem

Fig. 18. Impact of Buffer Sizes and Formats on Compression Performance and Ratios.

(3) **Transferred Size:** An important factor for cloud costs (and runtime performance because of network I/O) is the transferred data size, shown in Figures 18d (lineitem) and 18e (ICU). For lineitem, the columnar format, large buffers (512-1024KB), and zstd (followed by snappy, lzo, and lz4) lead to the smallest data size. However, these compression ratios and rankings are strongly data dependent. On ICU, zstd achieves 2× better compression while other libraries perform similar. Buffers larger than 256 KB and columnar format do not impact the data size.
(4) **Cloud-Edge:** Figure 18c shows the results for the ICU dataset in a cloud-edge environment. We assign the server 16 cores, the client 1, and limit the network to 10 MB/s. In this network-bound environment, the runtime largely depends on the transferred data size, and thus, zstd performs best, yielding almost an order of magnitude better end-to-end runtimes than uncompressed transfers. While we see that a columnar format does not improve performance for the ICU dataset, for the lineitem dataset (in Figure 18f) the columnar format with compression yields substantial improvements, especially for larger buffer sizes.
(5) **Compression Parallelism:** To analyze compression overhead and explore mitigation strategies, we transferred the lineitem dataset across unrestricted and restricted networks while increasing parallelism. In the unrestricted case (memory speed, Figure 19a), compression provides no performance improvement even with parallelism. At a parallelism degree of 1, compression causes slowdowns of up to 5× for zstd and ≈ 3× for snappy, but with 8 threads, the slowdown drops below 2×. In the restricted setting (1000 MB/s, Figure 19b), compression initially degrades performance by 2.5× (zstd) and 1.5× (snappy) for single-threaded execution. However, increasing parallelism mitigates this overhead, making compression effective; lightweight libraries (e.g., snappy, lz4, lzo) achieve up to 3× speedups with 8 threads. The performance degradation at 16 threads is due to global over-provisioning and context-switching overhead.

Insight: Compression is effective in restricted environments but requires careful tuning. Moderate buffers (256-512KB), lightweight compression (e.g., snappy), and columnar formats perform well for compressible datasets, while zstd and larger buffers reduce data size in extremely network-bound settings. Tuned compression parallelism (4-8 threads) mitigates runtime overhead.



(a) Local Copy (b) Copy over 1000 MB/s Network

Fig. 19. Cloud: Impact of Comp./Decomp. Parallelism.

Table 4. Simulated Use Case Environments.

Env	S.	C.	N.	Source	Target	Dataset
1: IoT	16	8	100	PostgreSQL	Pandas	IoT
2: Backup	32	16	1000	PostgreSQL	CSV	Lineitem
3: ICU	16	12	50	CSV	Pandas	ICU
4: Copy	8	8	∞	CSV	CSV	Lineitem
5: ETL	8	8	500	PostgreSQL	Spark	Lineitem
6: Migr.	8	8	100	PostgreSQL	PostgreSQL	Lineitem

Table 5. Optimization Times.

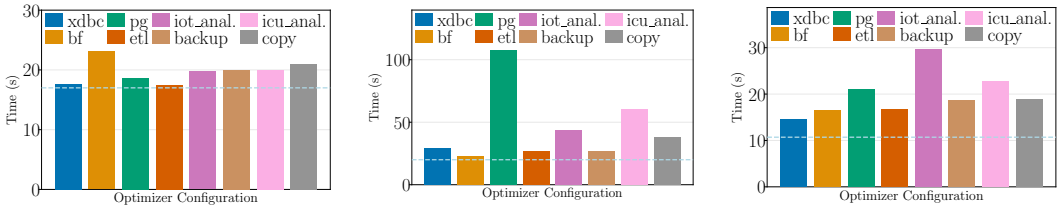
Env	Heuristic	Brute-force
1	66 ms	647 ms
2	**217 μs**	**30.5 s**
3	75 ms	1.7 s
4	191 μs	121 ms
5	73 ms	116 ms
6	**61 ms**	**3.2 s**

5.4 XDBC's Optimizer

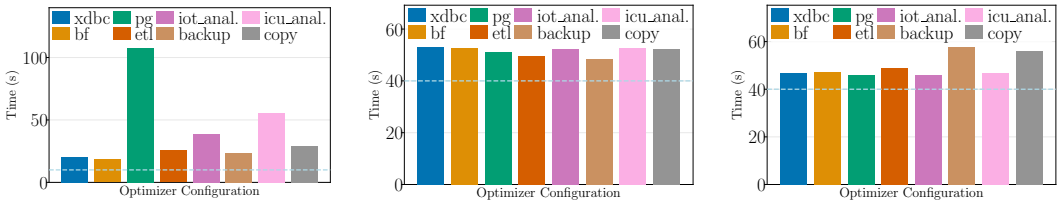
Having studied hand-crafted default configurations and systematic parameter sweeps, we now turn to the XDBC optimizer (see Section 4) for automatically finding good configurations. Given a set of performance characteristics obtained via offline profiling, a dataset, and a client/server environment (including networking), the optimizer provides the configuration that is expected to perform best, according to our cost model. Note that offline profiling only requires running the pipeline once, to obtain component throughputs.

Environments: Table 4 summarizes six different environments with different client/server cores, network limitations (where ∞ means local), source and target systems, as well as the datasets. These different environments represent realistic use cases with different hardware. For each environment, we manually designed an "expert configuration" for comparison.

Runtime Performance: Figure 20 shows the results for running the six expert configurations in all six use case environments. We mark a theoretical upper bound, typically determined by XDBC's slowest component, often the de/serializers. There are three major take aways. First, even for these well-chosen expert configurations, there is substantial variance in the runtimes. Some configurations like pg perform well in their environment (i.e., Env6) but perform the worst in others (e.g., Env2, Env4). Second, the expert configurations do not always yield the best performance in their intended environment. This result emphasizes the difficulty of chosen well-performing configurations, especially for non-expert users. Third, our optimizer finds robust configurations that lead to good runtime performance, which is on par with the expert configurations and in some cases outperforms all other configurations (e.g., Env 2 in Figure 20b and Env 3 in Figure 20c). We also implemented a brute-force optimizer (bf) that enumerates all valid configurations using the same cost model. Our heuristic optimizer is consistently close to, and occasionally outperforms, the brute-force approach.



(a) Env1 (IoT Analysis) (b) Env2 (Backup) (c) Env3 (ICU Analysis)



(d) Env4 (Copy) (e) Env5 (ETL) (f) Env6 (DB Migration)

Fig. 20. Optimizer configuration choices and impact. Dashed line marks the theoretically achievable runtime.

Cost Model Validation: To evaluate our used cost model, we examined the correlation between estimated and real throughput for the simulated environments. We show this correlation plot in Figure 21. We added the estimated costs for all manual configurations as well as the ones that the brute-force approach and heuristic optimizers produced, which we mark accordingly. While the cost is not accurately predicted, we can see that there is a correlation between estimated and real throughput, with most configurations being overestimated. This overestimation is expected due to the incremental worker assignment method employed by the optimizers. For instance, when a component's throughput is slightly below an upper bound, such as the available network bandwidth, the optimizer increases the worker count to surpass the bound, leading to an estimated throughput exceeding the actual network bandwidth. Additionally, manual configurations further amplify overestimation as their throughput calculations ignore bounds. For example, assigning many compression workers results in high estimated throughput, even when constrained by low network bandwidth.

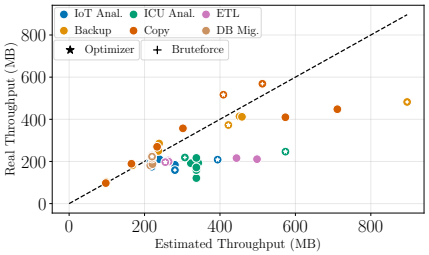


Fig. 21. Cost Model Analysis.

Optimization Time: Finally, Table 5 compares the optimization time of our heuristic optimizer with the brute-force optimizer (that explores the whole search space but prunes invalid configurations). While generally optimization is in the order of milliseconds, for some configurations the brute-force approach needs substantially more time to find the optimal configuration. In particular, for Env 2, which has 48 total workers, the optimizer needs 30*s* to produce an optimal configuration, while for Env 6 the optimizer needs 3*s*. This experiment shows that enumerating the configuration search space for large scale-up environments is prohibitive. With today's many-core environments, where 128 cores and more are not uncommon, the brute-force approach is not practical. Overall, our experiments show that our cost model is a solid basis for optimization and our heuristic optimizer robustly produces good plans in milliseconds.

Insight: The heuristic cost-based optimizer robustly and efficiently adapts configurations to diverse environments, delivering high performance with negligible optimization times.

6 Related Work

XDBC is broadly related to work on specialized connectors and data transfer techniques, ETL tools and integration platforms, as well as data exchange platforms and data transfer in the cloud.

Optimized Data Transfer: Apart from the already discussed data transfer optimizations, which suggest columnar intermediate formats [74] and leveraging parallelism [90], there is additional related work. Pipegen, the closest approach to ours, relies on unit tests for CSV I/O to automatically generate data transfer sockets [48], but lacks tuning knobs for individual transfer steps. Noll et al. further improved data loading with compressed storage, offloading compression to the client [69]. Other approaches include FastDAWG [91] (which leverages RDMA over infiniband to improve inter-DBMS data transfers), Muses [54] (which directly transfers intermediate data between distributed systems), and Portage [39] (which implements serializers/deserializers for every system pair, to avoid intermediate representations). Additionally, Apache Arrow Flight is an RPC framework (using gRPC and the IPC format) for data services on Arrow data [15]. It enables efficient transfers of Arrow data, but does not allow scaling or customizing individual components of the transfer pipeline. In contrast to specialized techniques, XDBC is a holistic, end-to-end data transfer framework with automatic optimization according to data and environment characteristics.

ETL Tools: There exist a variety of commercial platforms for extraction-transformation-loading (ETL) processes and data integration, including IBM Infosphere [4], Oracle GoldenGate [7], Informatica [5], Pentaho Kettle [9]. These ETL tools allow users to consolidate data from heterogeneous data sources through ETL processes into data warehouse environments, and offer rich functionality with many connectors and pipelines across multiple source and target systems, as well as visual workflow editors for composing these ETL flows. Earlier research also explored the optimization of such multi-system ETL pipelines [80, 81]. Additionally, Rheem's [21] optimizer considers data transfer costs for finding efficient cross-platform execution plans [60]. However, in contrast to XDBC, these works do not consider low-level details of data transfer and knob tuning for adapting to the environment. XDBC could be integrated in those platforms to provide more efficient data transfers between individual systems.

Integration Platforms: Besides ETL tools, enterprise application integration (EAI) aims to integrate arbitrary heterogeneous systems and applications. Examples are SAP Process Integration, IBM Message Broker (now IBM App Connect) [50], and Microsoft Biztalk Server (now Azure Integration Services) [66]. These systems often use persistent message queues, XML/JSON as a general but costly intermediate representation, (streaming) XML transformations and other operations, as well as a wide variety of inbound/output adapters/connectors for interacting with external systems. Example adapters include SQL and web-service sources, and workflows across such sources exhibit optimization opportunities [82, 88, 89]. In contrast, XDBC focuses on efficient data transfer and low-level tuning according to environment characteristics.

Efficient Readers and Processors: There has been extensive work on parsing and loading different file formats, including CSV [37, 45, 61, 68, 83], JSON [63, 65, 70], XML [33, 58, 62, 72], and spreadsheets [27, 43, 75]. Other work focused on directly processing external files without explicitly loading them in a database system [24, 32, 55]. All these approaches are orthogonal to XDBC. The first category can be leveraged as physical reader implementations in the XDBC Server for different file formats. The second category can be leveraged for directly processing external files from other systems, using the XDBC Client, without loading them in a DBMS.

Data Exchange Platforms: Furthermore, there has been extensive work on hybrid systems that leverage HDFS and parallel DBMSes (data warehouses). For example, Polybase proposes to interleave processing and storage capabilities of both data flow engines (e.g., MapReduce) and DBMSes (e.g., Microsoft PDW), by materializing data periodically, or directly streaming data

between systems for individual queries [36]. MISO proposes a cost-based optimizer to decide on physical design, i.e., which data to move where [64]. Extending the work on Polybase—which only pushed filters and joins for co-located data—Zigzag join proposes to exchange Bloom filters to enhance join processing across MapReduce and DBMSes [86]. While all these approaches improve the overall data transfer, their goal being to reduce data transfer, they do not focus on how to transfer data efficiently. Accordingly, XDBC could be employed for accelerating the remaining data transfer in such systems.

Data Transfer in the Cloud: Recently, there has been a shift towards analytics platforms in the cloud [59, 87]. Therefore, existing work aims to optimize data analytics workflows directly on cloud object storage [38]. While we do not directly consider data analytics, XDBC can be leveraged to accelerate data loading from different cloud providers and instances, considering the variety of hardware characteristics of different cloud instances. Additionally, to facilitate intra and inter-cloud transfers for ETL and geo-distributed analytics, cloud providers offer their own bulk transfer tools [1–3], at the time focusing on object storage systems. To optimize such transfers for runtime and monetary cost, Skyplane proposes to route bulk transfers across different regions [51]. While XDBC and Skyplane share common goals regarding runtime performance, the approaches are orthogonal to each other. Skyplane is mostly concerned with object storage data access and not heterogeneous systems and thus, could be extended with XDBC to support more sources and adaptively tune its routing algorithm.

7 Conclusions

Data transfer is a crucial task in today's decentralized data ecosystem. In this paper, we introduced XDBC as a holistic data transfer framework which achieves both good generality for easily connecting heterogeneous data systems, as well as performance close to specialized point-to-point connectors. We draw two major conclusions. First, the abstraction of different pipeline components with multiple configurable physical implementations provides great flexibility for tuning the data transfer pipeline according to environment and data characteristics. Second, our simple, rule-based, heuristic optimizer is able to quickly find very good configurations despite the large search space. Together, XDBC yields substantial runtime improvements compared to generic data transfer tools, sometimes even outperforming specialized connectors.

Interesting future work includes additional physical component primitives (e.g., RDMA for send/receive [52], computational storage for read/serialize [25]), integration into composable data systems [42, 71] (e.g., efficient data transfer between modular execution engines [40, 79]), as well as support for more general integration flows [28, 80, 81] in terms of workflows or data flows graphs with additional operations (e.g., two source systems, data enrichment, target system).

Acknowledgments

We gratefully acknowledge funding from the German Federal Ministry of Education and Research under the grants BIFOLD24B and 01IS17052 (as part of the Software Campus project PolyDB). We thank our anonymous reviewers for their constructive comments and suggestions, which helped improve the paper. We would like to thank the whole PolyDB team, and in particular Joel Ziegler, Midhun Kaippillil Venugopalan, and Benedikt Didrich, for their support in the ongoing development and integration of the prototype. We also thank the late Jorge-Arnulfo Quiané-Ruiz for insightful discussions during the early stages of this work.

References

[1] 2024. Amazon Web Services. Aws DataSync: online data transfer and migration. https://aws.amazon.com/datasync. Accessed: 2024-10-07.

[2] 2024. Azure storage AzCopy. https://github.com/Azure/azure-storage-azcopy. Accessed: 2024-10-07.

[3] 2024. Google Cloud Platform. Storage transfer service. https://cloud.google.com/storage-transfer-service. Accessed: 2024-10-07.

[4] 2024. IBM InfoSphere Information Server. https://www.ibm.com/information-server. Accessed: 2024-10-07.

[5] 2024. Informatica | Cloud Data Integration for Data Engineering. https://www.informatica.com/products/cloud-data-integration.html. Accessed: 2024-10-07.

[6] 2024. Java JDBC API. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/. Accessed: 2024-01-11.

[7] 2024. Oracle Goldengate | Replicate and Transform Data. https://www.oracle.com/integration/goldengate/. Accessed: 2024-10-07.

[8] 2024. Pandas API reference | pandas.read_sql. https://pandas.pydata.org/docs/reference/api/pandas.read_sql.html. Accessed: 2024-10-07.

[9] 2024. Pentaho Data Integration: Ingest, Blend, Orchestrate, and Transform Data. https://pentaho.com/products/pentaho-data-integration/. Accessed: 2024-10-07.

[10] 2024. PostgreSQL wiki | Foreign data wrappers. https://wiki.postgresql.org/wiki/Foreign_data_wrappers#Specific_SQL_Database_Wrappers. Accessed: 2024-10-07.

[11] 2024. Spark 3.5.3 Documentation | JDBC To Other Databases. https://spark.apache.org/docs/latest/sql-data-sources-jdbc.html. Accessed: 2024-10-07.

[12] 2024. Turbodbc - Turbocharged database access for data scientists. https://turbodbc.readthedocs.io/en/latest/. Accessed: 2024-10-07.

[13] 2025. American Community Survey Data. https://www.census.gov/programs-surveys/acs/data.html. Accessed: 2025-01-10.

[14] 2025. Apache Arrow. https://arrow.apache.org/. Accessed: 2025-01-10.

[15] 2025. Arrow Flight RPC. https://arrow.apache.org/docs/format/Flight.html. Accessed: 2025-01-10.

[16] 2025. Dataset of legitimate IoT data (VARIoT). https://www.data.gouv.fr/en/datasets/dataset-of-legitimate-iot-data/. Accessed: 2025-01-10.

[17] 2025. Docker Docs | Resource constraints. https://docs.docker.com/engine/containers/resource_constraints/. Accessed: 2025-01-10.

[18] 2025. Docker Traffic Control. https://github.com/lukaszlach/docker-tc. Accessed: 2025-01-10.

[19] 2025. JDBC Foreign Data Wrapper for PostgreSQL. https://github.com/pgspider/jdbc_fdw. Accessed: 2025-01-10.

[20] 2025. TPC BENCHMARK H. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf. Accessed: 2025-01-10.

[21] Divy Agrawal, Sanjay Chawla, Bertty Contreras-Rojas, Ahmed Elmagarmid, Yasser Idris, Zoi Kaoudi, Sebastian Kruse, Ji Lucas, Essam Mansour, Mourad Ouzzani, et al. 2018. RHEEM: enabling cross-platform data processing: may the big data be with you! *Proceedings of the VLDB Endowment* 11, 11 (2018), 1414–1427. https://doi.org/doi/10.14778/3236187.3236195

[22] Rakesh Agrawal and Kyuseok Shim. 1996. Developing Tightly-Coupled Data Mining Applications on a Relational Database System.. In *KDD*, Vol. 96. 287–290. https://cdn.aaai.org/KDD/1996/KDD96-049.pdf

[23] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. 2001. Weaving Relations for Cache Performance. In *Proc. VLDB Endow.* 169–180. https://www.vldb.org/conf/2001/P169.pdf

[24] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *SIGMOD*. 241–252. https://doi.org/10.1145/2213836.2213864

[25] Antonio Barbalace and Jaeyoung Do. 2021. Computational Storage: Where Are We Today?. In *CIDR*. http://cidrdb.org/cidr2021/papers/cidr2021_paper29.pdf

[26] Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2021. Compliant Geo-distributed Query Processing. In *SIGMOD '21: International Conference on Management of Data*. 181–193. https://doi.org/10.1145/3448016.3453687

[27] Mangesh Bendre, Bofan Sun, Ding Zhang, Xinyan Zhou, Kevin ChenChuan Chang, and Aditya Parameswaran. 2015. Dataspread: Unifying databases and spreadsheets. In *VLDB*, Vol. 8. 2000. https://doi.org/10.14778/2824032.2824121

[28] Matthias Boehm. 2011. *Cost-based optimization of integration flows.* Ph. D. Dissertation. Dresden University of Technology. https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa-67936

[29] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data Management in Machine Learning Systems.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00895ED1V01Y201901DTM057

[30] Gerard M. Campbell. 1991. Cyclical queueing systems. *European Journal of Operational Research* 51, 2 (1991), 155–167. https://doi.org/10.1016/0377-2217(91)90246-R

[31] Ankit Chaudhary, Kaustubh Beedkar, Jeyhun Karimov, Felix Lang, Steffen Zeuch, and Volker Markl. 2025. Incremental Stream Query Placement in Massively Distributed and Volatile Infrastructures. In *41st IEEE International Conference on*

Data Engineering ICDE.

[32] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *SIGMOD*. 1287–1298. https://doi.org/10.1145/2588555.2593673

[33] Suren Chilingaryan. 2009. The XMLBench Project: Comparison of Fast, Multi-platform XML libraries. 21–34. https://doi.org/10.1007/978-3-642-04205-8_4

[34] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. 2009. MAD Skills: New Analysis Practices for Big Data. *Proc. VLDB Endow.* 2, 2 (2009), 1481–1492. https://doi.org/10.14778/1687553.1687576

[35] Amol Deshpande, Joseph M. Hellerstein, and Vijayshankar Raman. 2006. Adaptive query processing: why, how, when, what next. In *SIGMOD*. 806–807. https://doi.org/10.1145/1142473.1142603

[36] David J. DeWitt, Alan Halverson, Rimma V. Nehme, Srinath Shankar, Josep Aguilar-Saborit, Artin Avanes, Miro Flasza, and Jim Gramling. 2013. Split query processing in polybase. In *SIGMOD*. ACM, 1255–1266. https://doi.org/10.1145/2463676.2463709

[37] Till Döhmen, Hannes Mühleisen, and Peter Boncz. 2017. Multi-hypothesis CSV parsing. In *SSDBM*. 1–12. https://doi.org/10.1145/3085504.3085520

[38] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782. https://doi.org/10.14778/3611479.3611486

[39] 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, 1–6. https://doi.org/10.1109/HPEC.2016.7761594

[40] Haralampos Gavriilidis. 2019. Computation offloading in jvm-based dataflow engines. In *BTW 2019–Workshopband*. 195–204. https://doi.org/doi:10.18420/btw2019-ws-20

[41] Haralampos Gavriilidis, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. In-Situ Cross-Database Query Processing. In *ICDE*. https://doi.org/10.1109/ICDE55515.2023.00214

[42] Haralampos Gavriilidis, Lennart Behme, Christian Munz, Varun Pandey, and Volker Markl. 2025. CompoDB: A Demonstration of Modular Data Systems in Practice. (2025). https://doi.org/10.48786/edbt.2025.97

[43] Haralampos Gavriilidis, Felix Henze, Eleni Tzirita Zacharatou, and Volker Markl. 2023. SheetReader: Efficient Specialized Spreadsheet Parsing. *Inf. Syst.* 115 (2023), 102183. https://doi.org/10.1016/J.IS.2023.102183

[44] Haralampos Gavriilidis, Leonhard Rose, Joel Ziegler, Kaustubh Beedkar, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. 2023. XDB in Action: Decentralized Cross-Database Query Processing for Black-Box DBMSes. *Proc. VLDB Endow.* 16, 12 (2023), 4078–4081. https://doi.org/10.14778/3611540.3611625

[45] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *SIGMOD*. 883–899. https://doi.org/10.1145/3299869.3319898

[46] John L. Gustafson. 1988. Reevaluating Amdahl's Law. *Commun. ACM* 31, 5 (1988), 532–533. https://doi.org/10.1145/42411.42415

[47] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. 2005. QPipe: A Simultaneously Pipelined Relational Query Engine. In *SIGMOD*. 383–394. https://doi.org/10.1145/1066157.1066201

[48] 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*. 470–483. https://doi.org/10.1145/2987550.2987567

[49] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[50] IBM 2024. *IBM App Connect*. IBM. https://www.ibm.com/support/knowledgecenter/en/SSTTDS_11.0.0/com.ibm.etools.mft.doc/ab20551_.htm.

[51] Paras Jain, Sam Kumar, Sarah Wooders, Shishir G. Patil, Joseph E. Gonzalez, and Ion Stoica. 2023. Skyplane: Optimizing Transfer Cost and Throughput Using Cloud-Aware Overlays. In *NSDI*. USENIX Association, 1375–1389. https://www.usenix.org/conference/nsdi23/presentation/jain

[52] Matthias Jasny, Lasse Thostrup, Sajjad Tamimi, Andreas Koch, Zsolt István, and Carsten Binnig. 2024. Zero-sided RDMA: Network-driven Data Shuffling for Disaggregated Heterogeneous Cloud DBMSs. *Proc. ACM Manag. Data* 2, 1 (2024), 36:1–36:28. https://doi.org/10.1145/3639291

[53] Alistair EW Johnson, Tom J Pollard, Lu Shen, Li-wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. 2016. MIMIC-III, a freely accessible critical care database. *Scientific data* 3, 1 (2016), 1–9. https://doi.org/10.1038/sdata.2016.35

[54] Abdulrahman Kaitoua, Tilmann Rabl, Asterios Katsifodimos, and Volker Markl. 2019. Muses: distributed data migration system for polystores. In *ICDE*. IEEE, 1602–1605. https://doi.org/10.1109/ICDE.2019.00152

[55] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. 2014. Adaptive query processing on RAW data. *PVLDB* 7, 12 (2014), 1119–1130. https://doi.org/10.14778/2732977.2732986

[56] Bettina Kemme and Gustavo Alonso. 2010. Database replication: a tale of research across communities. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 5–12. https://doi.org/10.14778/1920841.1920847
[57] Ernest Koenigsberg. 1982. Twenty Five Years of Cyclic Queues and Closed Queue Networks: A Review. *The Journal of the Operational Research Society* 33, 7 (1982), 605–619. http://www.jstor.org/stable/2581723
[58] Margaret G Kostoulas, Morris Matsa, Noah Mendelsohn, Eric Perkins, Abraham Heifets, and Martha Mercaldi. 2006. XML screamer: an integrated approach to high performance XML parsing, validation and deserialization. In *WWW*. 93–102. https://doi.org/10.1145/1135777.1135796
[59] Tim Kraska, Tianyu Li, Samuel Madden, Markos Markakis, Amadou Ngom, Ziniu Wu, and Geoffrey X Yu. 2023. Check out the big brain on BRAD: simplifying cloud data processing with learned automated data meshes. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3293–3301. https://doi.org/10.14778/3611479.3611526
[60] Sebastian Kruse, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, Sanjay Chawla, Felix Naumann, and Bertty Contreras-Rojas. 2019. Optimizing cross-platform data movement. In *ICDE*. IEEE, 1642–1645. https://doi.org/10.1109/ICDE.2019.00162
[61] Alexander Kumaigorodski, Clemens Lutz, and Volker Markl. 2021. Fast CSV Loading Using GPUs and RDMA for In-Memory Data Processing. *BTW* (2021). https://doi.org/10.18420/BTW2021-01
[62] Tak Cheung Lam, Jianxun Jason Ding, and Jyh-Charn Liu. 2008. XML document parsing: Operational and performance characteristics. *Computer* 41, 9 (2008), 30–37. https://doi.org/10.1109/MC.2008.403
[63] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960. https://doi.org/10.1007/S00778-019-00578-5
[64] Jeff LeFevre, Jagan Sankaranarayanan, Hakan Hacigümüs, Jun'ichi Tatemura, Neoklis Polyzotis, and Michael J. Carey. 2014. MISO: souping up big data query processing with a multistore system. In *SIGMOD*. ACM, 1591–1602. https://doi.org/10.1145/2588555.2588568
[65] Yinan Li, Nikos R Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: a fast JSON parser for data analytics. *PVLDB* 10, 10 (2017), 1118–1129. https://doi.org/10.14778/3115404.3115416
[66] Microsoft 2020. *MS BizTalk Server*. Microsoft. https://www.infosys.com/services/api-economy/documents/azure-integration-services.pdf.
[67] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. In *OSDI*. 561–577. https://www.usenix.org/conference/osdi18/presentation/nishihara
[68] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant loading for main memory databases. *PVLDB* 6, 14 (2013), 1702–1713. https://doi.org/10.14778/2556549.2556555
[69] Stefan Noll, Jens Teubner, Norman May, and Alexander Böhm. 2020. Shared Load (ing): Efficient Bulk Loading into Optimized Storage.. In *CIDR*. http://cidrdb.org/cidr2020/papers/p2-noll-cidr20.pdf
[70] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter before you parse: Faster analytics on raw data with sparser. *PVLDB* 11, 11 (2018), 1576–1589. https://doi.org/10.14778/3236187.3236207
[71] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The composable data management system manifesto. *Proceedings of the VLDB Endowment* 16, 10 (2023), 2679–2685. https://doi.org/10.14778/3603581.3603604
[72] Eric Perkins, Margaret Kostoulas, Abraham Heifets, Morris Matsa, and Noah Mendelsohn. 2005. Performance analysis of XML APIs. Citeseer.
[73] Devin Petersohn, William W. Ma, Doris Jung Lin Lee, Stephen Macke, Doris Xin, Xiangxi Mo, Joseph Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya G. Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proc. VLDB Endow.* 13, 11 (2020), 2033–2046. https://doi.org/10.14778/3407790.3407807
[74] 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 (2017), 1022–1033. https://doi.org/10.14778/3115404.3115408
[75] Sajjadur Rahman, Kelly Mack, Mangesh Bendre, Ruilin Zhang, Karrie Karahalios, and Aditya Parameswaran. 2020. Benchmarking Spreadsheet Systems. In *SIGMOD*. 1589–1599. https://doi.org/10.1145/3318464.3389782
[76] M Tork Roth, Manish Arya, L Haas, Michael Carey, William Cody, Ronald Fagin, P Schwarz, Joachim Thomas, and E Wimmers. 1996. The garlic project. In *SIGMOD*. 557. https://doi.org/10.1145/233269.280363
[77] Sunita Sarawagi, Shiby Thomas, and Rakesh Agrawal. 1998. Integrating association rule mining with relational database systems: Alternatives and implications. *SIGMOD Record* 27, 2 (1998), 343–354. https://doi.org/10.1023/A:1009887712954
[78] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, et al. 2019. Presto: SQL on everything. In *ICDE*. IEEE, 1802–1813. https://doi.org/10.1109/ICDE.2019.00196
[79] Akash Shankaran, George Gu, Weiting Chen, Binwei Yang, Chidamber Kulkarni, Mark Rambacher, Nesime Tatbul, and David E Cohen. 2023. The Gluten Open-Source Software Project: Modernizing Java-based Query Engines for the Lakehouse Era.. In *VLDB Workshops*. https://ceur-ws.org/Vol-3462/CDMS8.pdf

[80] Alkis Simitsis, Panos Vassiliadis, and Timos K. Sellis. 2005. Optimizing ETL Processes in Data Warehouses. In *ICDE*. 564–575. https://doi.org/10.1109/ICDE.2005.103

[81] Alkis Simitsis, Kevin Wilkinson, and Petar Jovanovic. 2013. xPAD: a platform for analytic data flows. In *SIGMOD*. 1109–1112. https://doi.org/10.1145/2463676.2465247

[82] Utkarsh Srivastava, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. 2006. Query Optimization over Web Services. In *VLDB*. 355–366. http://dl.acm.org/citation.cfm?id=1164159

[83] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively Parallel Parsing of Delimiter-Separated Raw Data. *PVLDB* 13, 5 (2020), 616–628. https://doi.org/10.14778/3377369.3377372

[84] Michael Stonebraker and Uğur Çetintemel. 2018. " One size fits all" an idea whose time has come and gone. In *Making databases work: the pragmatic wisdom of Michael Stonebraker*. 441–462. https://doi.org/doi/10.1145/3226595.3226636

[85] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*. 1150–1160. http://www.vldb.org/conf/2007/papers/industrial/p1150-stonebraker.pdf

[86] Yuanyuan Tian, Fatma Özcan, Tao Zou, Romulo Goncalves, and Hamid Pirahesh. 2016. Building a Hybrid Warehouse: Efficient Joins between Data Stored in HDFS and Enterprise Warehouse. *ACM Trans. Database Syst.* 41, 4 (2016), 21:1–21:38. https://doi.org/10.1145/2972950

[87] Alexander van Renen and Viktor Leis. 2023. Cloud Analytics Benchmark. *Proc. VLDB Endow.* 16, 6 (2023), 1413–1425. https://doi.org/10.14778/3583140.3583156

[88] Marko Vrhovnik, Holger Schwarz, Sylvia Radeschütz, and Bernhard Mitschang. 2008. An Overview of SQL Support in Workflow Products. In *ICDE*. 1287–1296. https://doi.org/10.1109/ICDE.2008.4497538

[89] Marko Vrhovnik, Holger Schwarz, Oliver Suhre, Bernhard Mitschang, Volker Markl, Albert Maier, and Tobias Kraft. 2007. An Approach to Optimize Data Processing in Business Processes. In *VLDB*. 615–626. http://www.vldb.org/conf/2007/papers/research/p615-vrhovnik.pdf

[90] Xiaoying Wang, Weiyuan Wu, Jinze Wu, Yizhou Chen, Nick Zrymiak, Changbo Qu, Lampros Flokas, George Chow, Jiannan Wang, Tianzheng Wang, et al. 2022. ConnectorX: accelerating data loading from databases to dataframes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2994–3003. https://doi.org/10.14778/3551793.3551847

[91] 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: VLDB 2018 Workshops, Poly and DMAH, Rio de Janeiro, Brazil, August 31, 2018, Revised Selected Papers 4*. Springer, 3–15. https://doi.org/10.1007/978-3-030-14177-6_1

[92] Steffen Zeuch, Ankit Chaudhary, Bonaventura Del Monte, Haralampos Gavriilidis, Dimitrios Giouroukis, Philipp M Grulich, Sebastian Breß, Jonas Traub, and Volker Markl. 2020. The NebulaStream Platform: Data and Application Management for the Internet of Things. In *CIDR*. http://cidrdb.org/cidr2020/papers/p7-zeuch-cidr20.pdf

Received October 2024; revised January 2025; accepted February 2025